PLANNING WITH CONTINUOUS RESOURCES IN AGENT SYSTEMS

by

Janusz Marecki

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2008

Acknowledgments

First and foremost, I would like to thank my advisor, Milind Tambe, for the support and mentoring he has always been so enthusiastic to provide. I also want to thank the members of my thesis committee, Victor Lesser, Jonathan Gratch, Fernando Ordez and Rajiv Maheswaran for helping me to advance my research in the right direction and to think beyond it. Furthermore, I would like to thank Sven Koenig, Makoto Yokoo, Paul Scerri, Sarit Kraus and David Kempe for valuable insights on the material in this thesis and useful career guidance. Also, special thanks to Honeywell Inc. and the CREATE research center at USC for the support of this work.

I also want to thank my collegues at USC: I thank Nathan Schurr for the phenomenal job he did in introducing me to the American culture, Zvi Topol for our fabulous movie making endeavors, Jonathan Pearce for enthusiastically answering my mac-related questions, Pradeep Varakantham for showing me the concept of indoor cricket, Tapana Gupta and Manish Jain for teaching me Hindi, Praveen Paruchuri and Ranjit Nair for showing me how not to worry, Emma Bowring for revealing to me the secret of sweet-life-goals, William Yeoh for showing me how to be generous, Xiaosun Sun and Xiaoming Zheng for discussing with me the world issues, Chris Portway for his willingness to tolerate my accent and James Pita for motivating me to do the workout once in a while.

And finally, I would like to thank my family for never loosing hope in me during this graduate school adventure. Especially, I thank my wife Dorota for being there for me during this time..

Table of Contents

Acknowledgments ii

List Of Figures vii

Abstract ix

Chapter 1: Introduction 1
- 1.1 Assumptions . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Thesis Contributions . . . 4
- 1.4 Overview of Thesis . . . 6

Chapter 2: Motivating Domains and Formal Models 7
- 2.1 Motivating Domains . . . 7
  - 2.1.1 Planetary Exploration: Domain for Single Agent Algorithms . . . 8
  - 2.1.2 Adjustable Autonomy: Domain for Single Agent Algorithms . . . 9
  - 2.1.3 Civilian Rescue: Domain for Multiagent Algorithms . . . 11
    - 2.1.3.1 Example 1: Fully Ordered Domain . . . 12
    - 2.1.3.2 Example 2: Partially Ordered Domain . . . 14
  - 2.1.4 General Description of Multiagent Domains . . . 16
- 2.2 Formal Models . . . 18
  - 2.2.1 Markov Decision Process . . . 19
  - 2.2.2 Continuous Resource MDP . . . 20
  - 2.2.3 Continuous Resource, Decentralized MDP . . . 24

Chapter 3: Single Agent Solutions 29
- 3.1 Value Iteration Approach: The CPH Algorithm . . . 29
  - 3.1.1 CPH Step 1: Phase Type Approximation . . . 31
  - 3.1.2 CPH Step 2: Piecewise Gamma Value Functions . . . 34
  - 3.1.3 CPH Step 3: Functional Value Iteration . . . 36
    - 3.1.3.1 Simplified Example . . . 36
    - 3.1.3.2 The General Case . . . 39
  - 3.1.4 Error Control . . . 46
- 3.2 Forward Search Approach: The DPFP Algorithm . . . 50
  - 3.2.1 DPFP at a Glance . . . 50

3.2.2 Dual Problem . . . 51
3.2.3 Solving the Dual Problem . . . 54
3.2.4 Taming the Algorithm Complexity . . . 59
3.2.5 Error Control . . . 60

Chapter 4: Experiments with Single Agent Algorithms 63
4.1 CPH Feasibility Experiments . . . 63
4.2 CPH and DPFP Scalability Experiments . . . 67
4.3 DPFP-CPH Hybrid Experiments . . . 69
4.4 DEFACTO Experiments . . . 72

Chapter 5: Multiagent Solutions 78
5.1 Locally Optimal Solution: The VFP Algorithm . . . 78
5.1.1 Policy Iteration Approach . . . 81
5.1.2 Functional Representation . . . 83
5.1.3 Opportunity Cost Function Propagation Phase . . . 85
5.1.4 Probability Function Propagation Phase . . . 89
5.1.5 Splitting the Opportunity Cost Functions . . . 92
5.1.6 Implementation of Function Operations . . . 99
5.2 Globally Optimal Solution: The M-DPFP Algorithm . . . 100
5.2.1 Arriving at M-DPFP . . . 100
5.2.2 The M-DPFP Algorithm . . . 105
5.2.3 Representative States . . . 110
5.2.3.1 Representative State Selection . . . 110
5.2.3.2 Policy for non-Representative States . . . 115
5.2.3.3 Error Bound . . . 116
5.2.4 Lookahead Function . . . 120
5.2.4.1 Dual Problem Formulation . . . 121
5.2.4.2 Dual Problem and the Lookahead Function . . . 125
5.2.4.3 Solving the Dual Problem . . . 129

Chapter 6: Experiments with Multiagent Algorithms 131
6.1 VFP Experiments . . . 131
6.1.1 Evaluation of the Opportunity Cost Splitting Heuristics . . . 132
6.1.2 Comparison of VFP and OC-DEC-MDP Efficiency . . . 134
6.2 M-DPFP Experiments . . . 138
6.2.1 Efficiency of the M-DPFP Lookahead Function . . . 138
6.2.2 Efficiency of the M-DPFP algorithm . . . 140

Chapter 7: Related Work 143
7.1 Single Agent Systems . . . 143
7.1.1 Constrained MDPs . . . 143
7.1.2 Semi MDPs . . . 144
7.1.3 Continuous state MDPs . . . 145
7.2 Multiagent Systems . . . 148
7.2.1 Globally Optimal Solutions . . . 148

7.2.2 Locally Optimal Solutions 149

Chapter 8: Conclusions **151**
8.1 Summary 151
8.2 Future Work 153

Reference List **155**

Appendix A: Phase Type Distribution **160**

Appendix B: Classes of Phase Type Distributions **162**

Appendix C: Fitting to Phase-Type Distributions **165**

Appendix D: Uniformization of Phase Type Distributions **167**

List Of Figures

2.1 Simplified planetary exploration domain 9

2.2 DEFACTO disaster simulation system 10

2.3 Civilian rescue domain and a mission plan 13

2.4 Partially ordered multiagent domain 15

2.5 Agent interacting with the environment in a Markov Decision Process 19

3.1 Phase type approximation example 32

3.2 CPH Value function 35

3.3 Proof by induction of CPH value iteration 42

3.4 Forward search of DPFP 56

4.1 Empirical evaluation of CPH for exponential distributions. 66

4.2 Empirical evaluation of CPH for Weibull distributions. 67

4.3 Planning horizon of CPH. 67

4.4 Domains for CPH and DPFP scalability experiments 68

4.5 Scalability experiments for DPFP, CPH and Lazy Approximation 70

4.6 DPFP+CPH hybrid: Fully ordered domain 71

4.7 DPFP+CPH hybrid: Unordered domain 72

4.8 DPFP+CPH hybrid: Partially ordered domain 72

4.9 RIAACT: Adjustable autonomy component of the DEFACTO system 75

4.10 RIAACT results 77

5.1 Agent policy in a fully-ordered CR-DEC-MDPs 80

5.2 Method time window and execution window 86

5.3 Propagation of opportunity cost functions and probability functions 87

5.4 Visualization of the operation performed by Equation 5.3 88

5.5 Splitting the opportunity cost functions 93

5.6 M-DPFP algorithm: Generation of the representative states 101

5.7 Example representative states 102

5.8 Method sequences graph 108

5.9 Demonstration of the H_{UP} heuristic 114

5.10 Greedy policy for the non-representative states 115

5.11 Greedy policy and the error bound 117

5.12 The range of the lookahead function 126

6.1 Visualization of the opportunity costs splitting heuristics 133

6.2 Mission plan configurations 134

6.3 VFP performance in the civilian rescue domain. 135

6.4 Scalability experiments for OC-DEC-MDP and VFP 136

6.5 Runtime and quality of the lookahead function 139

6.6 Error bound of the of the Lookahead function 140

6.7 Runtime of the M-DPFP algorithm 141

6.8 Quality loss of the M-DPFP algorithm 142

Abstract

My research concentrates on developing reasoning techniques for intelligent, autonomous agent systems. In particular, I focus on planning techniques for both single and multi-agent systems acting in uncertain domains. In modeling these domains, I consider two types of uncertainty: (i) the outcomes of agent actions are uncertain and (ii) the amount of resources consumed by agent actions is uncertain and only characterized by continuous probability density functions. Such rich domains, that range from the Mars rover exploration to the unmanned aerial surveillance to the automated disaster rescue operations are commonly modeled as continuous resource Markov decision processes (MDPs) that can then be solved in order to construct policies for agents acting in these domains.

This thesis addresses two major unresolved problems in continuous resource MDPs. First, they are very difficult to solve and existing algorithms are either fast, but make additional restrictive assumptions about the model, or do not introduce these assumptions but are very inefficient. Second, continuous resource MDP framework is not directly applicable to multi-agent systems and current approaches all discretize resource levels or assume deterministic resource consumption which automatically invalidates the formal solution quality guarantees. The goal of my thesis is to fundamentally alter this landscape in three contributions:

I first introduce CPH, a fast analytic algorithm for solving continuous resource MDPs. CPH solves the planning problems at hand by first approximating with a desired accuracy the probability distributions over the resource consumptions with phase-type distributions, which use exponential distributions as building blocks. It then uses value iteration to solve the resulting MDPs more efficiently than its closest competitor, and allows for a systematic trade-off of solution quality for speed.

Second, to improve the anytime performance of CPH and other continuous resource MDP solvers I introduce the DPFP algorithm. Rather than using value iteration to solve the problem at hand, DPFP performs a forward search in the corresponding dual space of cumulative distribution functions. In doing so, DPFP discriminates in its policy generation effort providing only approximate policies for regions of the state-space reachable with low probability yet it bounds the error that such approximation entails.

Third, I introduce CR-DEC-MDP, a framework for planning with continuous resources in multi-agent systems and propose two algorithms for solving CR-DEC-MDPs: The first algorithm (VFP) emphasizes scalability. It performs a series of policy iterations in order to quickly find a locally optimal policy. In contrast, the second algorithm (M-DPFP) stresses optimality; it allows for a systematic trade-off of solution quality for speed by using the concept of DPFP in a multi-agent setting.

My results show up to three orders of magnitude speedups in solving single agent planning problems and up to one order of magnitude speedup in solving multi-agent planning problems. Furthermore, I demonstrate the practical use of one of my algorithms in a large-scale disaster simulation where it allows for a more efficient rescue operation.

Chapter 1: Introduction

Recent years have seen an unprecedented rise in interest in the deployment of aerial [Beard and McLain, 2003], underwater [Blidberg, 2001] and terrestrial [Bresina et al., 2002; Thrun et al., 2006] unmanned vehicles to perform a variety of tasks in environments that are often hazardous and inaccessible to humans. Whereas some of these vehicles are tele-operated [Beard and McLain, 2003], vehicles such as the Mars rovers must often act autonomously due to inherent communication limitations with their human operators [Bresina et al., 2002]. As a result, automated planning techniques for these vehicles have recently received a lot of attention [Pedersen et al., 2005].

Of particular importance are planning techniques that take into account the uncertain nature of agent environments, to construct robust plans for all possible agent action outcomes, including action failures. For example, current Mars rovers receive only high-level instruction such as to "move from site A to site B" and plan the optimal traversal between these sites themselves. In doing so, they currently do not plan for action failures. As a result, it has been estimated [Mausam et al., 2005] that the 1997 Mars Pathnder rover spent between 40% and 75% of its time doing nothing because plans did not execute as expected.

Furthermore, planning techniques for autonomous agents must often take into account the energy requirements of these agents, or more generally, resource requirements for resources such as time, energy, storage whose consumption is uncertain and can only characterized by continuous probability density functions. For example, with the data collected during the 1997 Mars Pathfinder mission, NASA researchers [Bresina et al., 2002] are now in possession of statistical distributions of energy and time required to perform various rover activities. That abundance of data has in turn encouraged the Artificial Intelligence community to develop more expressive planning frameworks and more efficient algorithms for planning in continuous, dynamic and uncertain environments [Pedersen et al., 2005].

1.1 Assumptions

Planning problems for autonomous agents are often characterized by different assumptions. In particular, the planning problems that this thesis considers share the following characteristics:

- Agents can fully observe their local states and know the rewards for their actions, assumed to be are positive and obtained upon transitioning to new states.
- The execution of agent actions is related to resource availability: Initial resource levels are known, actions cause resource levels to decrease (according to a given continuous probability density function) and can be executed only if required resources are available. This thesis considers a case where agents have to deal with a single type of resource.
- The goal of the agents is to maximize reward yields.

1.2 Problem Statement

Continuous resource MDPs have emerged as a powerful planning technique to address such domains with uncertain resource consumption and resource limits. Unfortunately, continuous resource MDPs are very difficult to solve, both for single and multi-agent settings.

For single agent settings, existing algorithms either (i) make additional restrictive assumptions about the model or (ii) do not add new assumptions, but run very slow. The first limitation applies to Constraint MDPs [Altman, 1999] [Dolgov and Durfee, 2005] that do not allow for stochastic resource consumption or Semi MDPs and Generalized Semi MDPs [Puterman, 1994] [Younes and Simmons, 2004] that do not allow policies to be indexed by the amount of resources left. On the other hand, the second limitation applies to existing value iteration algorithms [Boyan and Littman, 2000], [Feng et al., 2004], [Liu and Koenig, 2005] and [Li and Littman, 2005] or policy iteration algorithms [Lagoudakis and Parr, 2003], [Hauskrecht and Kveton, 2004], [Nikovski and Brand, 2003], [Dolgov and Durfee, 2006], [Petrik, 2007] [Mahadevan and Maggioni, 2007] which may exhibit poor performance with the scale-up in their state spaces. To remedy that, [Guestrin et al., 2004] developed a technique to factor the underlying MDP thus reducing its complexity and [Mausam et al., 2005] developed a technique that prunes out the unreachable states and considers the remaining states in the order of their importance. Unfortunately, even with these improvements, current algorithms for solving continuous resource MDPs may still run slow in domains where solution quality guarantees are required [Pedersen et al., 2005].

For multi-agent settings, the situation is even worse, as continuous resource MDPs have received little attention, despite the increasing popularity of multi-agent domains with continuous characteristics [Raja and Lesser, 2003], [Becker et al., 2003], [Lesser et al., 2004], [Musliner

et al., 2006]. In fact, the proposed globally optimal algorithms [Becker et al., 2003], [Becker et al., 2004], [Nair et al., 2005], [Varakantham et al., 2007] as well as locally optimal algorithms [Nair et al., 2003], [Beynier and Mouaddib, 2005], [Beynier and Mouaddib, 2006] all discretize the continuous resources and encode them as separate features of the state. As a consequence, the planning horizons of those algorithms are limited to only a few time ticks. On top of that, the discretization process invalidates the formal solution quality guarantees that these algorithms provide.

Thus, there are two unresolved problems to be addressed: The first is to design new efficient algorithms for continuous resource MDPs that are faster than existing algorithms, while providing error bounds. The second is to design a framework and algorithms for planning with continuous resources in multi-agent systems.

1.3 Thesis Contributions

In this context I introduce the three major contributions of my thesis: First, I develop a new method for solving MDPs with continuous resources called CPH (**C**ontinuous resource MDPs through **PH**ase-type distributions) which finds solutions with quality guarantees, but is several orders of magnitude faster than its closest competitor, the Lazy Approximation algorithm [Li and Littman, 2005]. CPH approximates the probability distributions with phase-type distributions, which use exponential probability distributions as building blocks [Neuts, 1981]. It then uses the analytical value iteration technique to solve the resulting MDPs by exploiting the properties of exponential probability distributions to derive the necessary convolutions efficiently and precisely. Furthermore, I demonstrate the successful integration of CPH with RIAACT (**R**easoning

in **A**djustable **A**utonomy in **C**ontinuous **T**ime) [Schurr et al., 2008], the adjustable autonomy component of the DEFACTO system [Schurr et al., 2005] for training the incident commanders of the Los Angeles Fire Department.

Second, I introduce a DPFP algorithm (**D**ynamic **P**robability **F**unction **P**ropagation), to improve the anytime performance of CPH and other techniques for solving MDPs with continuous resources. Rather than performing value iteration to solve the problem at hand, DPFP performs a forward search in the corresponding space of cumulative distribution functions. In doing so, DPFP discriminates in its policy generation effort providing only approximate policies for regions of the state-space reachable with low probability yet it is able to bound the error that such approximation entails. When run alone, DPFP outperforms other algorithms in terms of its anytime performance, whereas when run as a hybrid, it allows for a significant speedup of CPH.

The third contribution in my thesis provides techniques for planning with continuous resources in a multi-agent setting. To this end I introduce **C**ontinuous **R**esource, **Dec**entralized MDP (CR-DEC-MDP), a first framework for planning with continuous resources in multi-agent systems and propose two algorithms for solving problems modeled as CR-DEC-MDP: **V**alue **F**unction **P**ropagation (VFP) and **M**ultiagent DPFP (M-DPFP). VFP emphasizes scalability; it leverages the value function reasoning to a multi-agent setting, that is, it maintains and manipulates a value function over time for each state rather than a separate value for each pair of method and time interval. Such representation allows VFP to group the time points for which the value function changes at the same rate which gives a one order of magnitude speedup over VFP's closest competitor, the OC-DEC-MDP algorithm [Beynier and Mouaddib, 2005]. In addition, VFP identifies and corrects critical overestimations of the expected value of a policy, that OC-DEC-MDP fails to address. On the other hand, in order to solve CR-DEC-MDPs optimally, I propose

the M-DPFP algorithm. M-DPFP finds policies with solution quality guarantees by employing the concept of forward search in the space of cumulative distribution functions in a multi-agent setting.

1.4 Overview of Thesis

Having outlined the major theme of my thesis, the rest of my thesis document is organized as follows: Chapter 2 provides the motivating domains for my work and introduces the formal frameworks for planning with continuous resources: (i) The Continuous Resource MDP model for single agent systems and (ii) the Continuous Resource, Decentralized MDP model for multiagent systems. In Chapter 3, that focuses on a single agent case, I develop two algorithms for solving problems modeled as continuous resource MDPs: A value iteration algorithm (CPH) that attains high quality solutions and a forward search algorithm (DPFP) that exhibits superior anytime performance. The experimental evaluation of CPH, DPFP as well as a DPFP-CPH hybrid algorithm that combines the strengths of the two algorithms is presented in Chapter 4. Furthermore, Chapter 4 also reports on the successful integration of CPH with the DEFACTO disaster rescue simulation system. I then move on to multiagent planning problems. In Chapter 5 I develop two algorithms for solving problems modeled as CR-DEC-MDPs: A fast, locally optimal algorithm (VFP) and a globally optimal algorithm (M-DPFP). The experimental study of these algorithms is presented in Chapter 6. I discuss the related work in Chapter 7. The summary of my dissertation and future research directions are outlined in Chapter 8.

Chapter 2: Motivating Domains and Formal Models

In this chapter I first introduce three different domains that motivate this research: (i) A planetary exploration domain for single agent algorithms, (ii) a disaster rescue domain for single agent algorithms and (iii) a civilian rescue domain for multiagent algorithms. Next, I provide a general description of the class of multiagent domain that this thesis focuses on. Finally, for modeling the planning problems of interest, I introduce two formal frameworks: (i) Continuous Resource MDP, for single agent planning problems and (ii) Continuous Resource, Decentralized MDP, for multiagent planning problems.

2.1 Motivating Domains

This section presents three different domains that motivate this research: Section 2.1.1 introduces a planetary exploration domain for single agent algorithms, Section 2.1.2 introduces a disaster rescue domain for single agent algorithms and finally, Section 2.1.3.1 introduces a civilian rescue domain for multiagent algorithms. Finally, Section 2.1.4 provides a general description of the class of multiagent domain that this thesis focuses on.

2.1.1 Planetary Exploration: Domain for Single Agent Algorithms

I consider time as an example continuous resource and focus on a simplified version of the planetary exploration problem originally introduced by [Bresina et al., 2002] and later extended by [Pedersen et al., 2005] as a motivation for my work. Following the success of the first NASA Mars exploration rover *Sojourner*, there are currently two other rovers rolling on the surface of Mars: *Spirit* and *Opportunity*. The mission of each rover usually consists of visiting various exploration sites, and transferring the collected data back to the lander base that can later relay it back to Earth. There are two major challenges that rover exploration presents: (i) since it takes 20 minutes for a control signal from Earth to reach Mars, rovers must act autonomously, and (ii) because of the uncertain nature of the terrain the rovers traverse, their actions are not deterministic. For example, is has been estimated that *Sojourner* spent between 40% and 75% of its time doing nothing because the execution of its actions resulted in situations that have not been planned for, and consequently the rover had to wait for the arrival of new commands from the mission operation center on Earth. Since rovers have limited life-span, is it crucial to collect the important data as soon as possible while ensuring that a rover does not get lost.

Exploration rovers rely on the solar power, and can only operate during a day. At each time of rover operation hours, one can estimate the remaining time of rover operation for that day. For simplicity, this remaining time of operation for a day will be referred to as time-to-deadline. Figure 2.1 illustrates a simple version of the problem outlined by [Bresina et al., 2002]. A rover has to maximize its expected total reward with time-to-deadline $\Delta = 4$. There are five locations that the rover can be in: Rover start location, site 1, site 2, site 3 and rover base station. The rover can perform two actions with deterministic action outcomes outside of its base station: (1)

It can move to the *next* site and collect a rock probe from there. It receives rewards 4, 2 and 1 for collecting rock probes from sites 1, 2 and 3, respectively. (2) It can also skip all remaining sites and move back to its base station to perform a communication task, which drains its energy completely and thus makes it impossible for the rover to perform additional actions. It receives reward 6 for performing the communication task. Finally, action durations are not deterministic, they are sampled from either exponential, normal or Weibull distributions. For a domain in Figure 2.1 the ordering in which the rover has to visit different sites is already given. However, depending on the amount of time left, the rover does not know whether it should (i) move to the next site or (ii) return to base. Hence, for any point in time and any site of interest, the rover must be able to efficiently determine the total expected utilities of these two different actions.

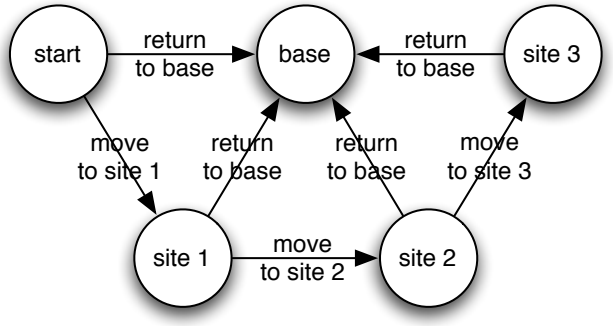

Figure 2.1: Simplified planetary exploration domain

2.1.2 Adjustable Autonomy: Domain for Single Agent Algorithms

Another important domain that motivates this thesis is the adjustable autonomy problem for a disaster rescue domain shown in Figure 2.2. In this domain a large scale disaster such as an earthquake hits a city, resulting in multiple buildings going on fire at the same time. With little

or no time to prepare for such an eventuality, a disaster rescue operation has to be performed in a coordinated fashion. In particular, one can envision an incident commander (a person in Figure 2.2) who makes strategic decisions about which fires to fight first. For example, an incident commander might be aware of the fact that the overall damage in the city can be reduced if some areas of the city are sacrificed in favor of other areas. The incident commander can then summon all the available resources (fire engines) to perform a particular task rather than having the resources be used inefficiently in a non-coordinated fashion on multiple other tasks.

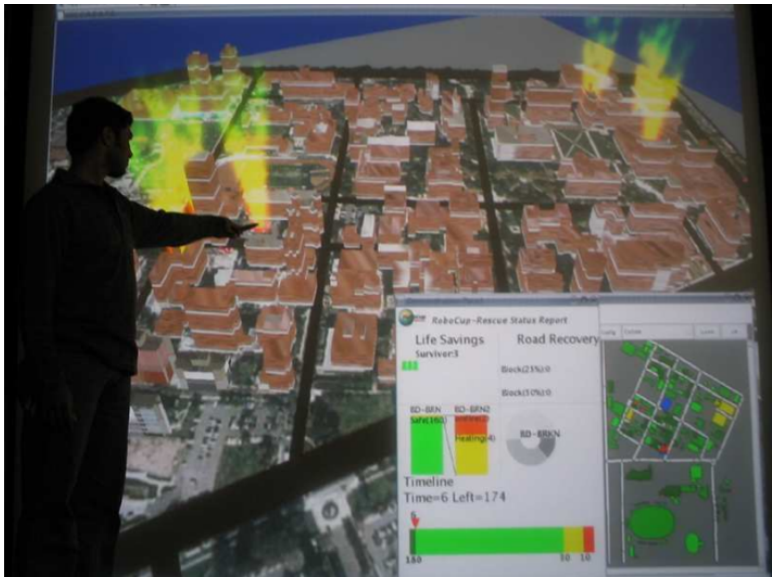

Figure 2.2: DEFACTO disaster simulation system

However, the human incident commander can quickly become overwhelmed by the number of role allocation requests from the fire engines participating in the disaster rescue operation [Marecki et al., 2005]. To alleviate that, some of the role allocation burden needs to be shifted from the human incident commander to a fire engine agent [Schurr et al., 2005]. In essence, the

agent faces time as a continuous resource to determine whether to take advice from human or not, and when advice is obtained and the agent disagrees with it, whether to continue negotiating with a human or not.

2.1.3 Civilian Rescue: Domain for Multiagent Algorithms

Planning under uncertainty for agent teams is more challenging than for single agents systems, because of the lack of global state knowledge in multiagent systems. In essence, even though an agent has incomplete knowledge of other agents' execution status, it must make optimal actions. The restriction about the lack of global state knowledge can in theory be eliminated by assuming that the agents can directly communicate. However, such an assumption introduces two problematic issues:

- If communication fails yet the agents are still executing policies that assume that reliable communication, the quality of such policies is questionable.

- Communication does not reduce the complexity of the problem. Indeed, as has been shown in [Nair et al., 2004], communication requires computationally intensive offline planning about all possible communication messages which often prevents the underlying algorithms from scaling up to big domains.

This thesis considers a communication mode that mitigates the above-mentioned issues, yet still allows the agents to exchange the information during the execution phase. What makes it possible is the use of the environment to transmit the information [Becker et al., 2004].

2.1.3.1 Example 1: Fully Ordered Domain

For the multi-agent domains in this thesis it is assumed that time is a continuous resource. In that context, agents must coordinate their plans over time, despite uncertainty in plan execution duration and outcome. In particular, this section introduces a fully-ordered domain where agents know the activities that they have to perform, yet do not know when these activities need to be performed. One example domain is large-scale disaster, like a fire in a skyscraper. Because there can be hundreds of civilians scattered across numerous floors, multiple rescue teams have to be dispatched, and radio communication channels can quickly get saturated and useless. In particular, firefighters must be sent on separate missions to rescue the civilians trapped in dozens of different locations.

Picture a small *mission plan* from Figure 2.3, where three firefighters have been assigned a task to rescue the civilians trapped at site B, accessed from site A. One example of such situation might be when site B is a conference room and site A is a hall. General fire fighting procedures involve both: (i) putting out the flames, and (ii) ventilating the site to let the toxic, high temperature gases escape, with the restriction that ventilation should not be performed too fast in order to prevent the fire from spreading. The team stars its mission at time $EST = 0$ (Earliest Starting Time) and estimates that at $LET = 20$ (Last Execution Time) the fire at site B will become unbearable for the civilians. In addition, the team knows that the fire at site A has to be put out in order to open the access to site B. Furthermore, each team member knows the actions (methods) that it has to execute (white boxes inside gray rectangles in Figure 2.3) and the ordering of its actions (dotted arrows in Figure 2.3).

As has happened in the past in large scale disasters, communication often breaks down and hence, the civilian rescue domain introduced here assumes that there is no explicit communication between firefighters 1,2 and 3 (denoted as FF1, FF2 and FF3). In particular, FF2 does not know if it is already safe to ventilate site A, FF1 does not know if site A has already been ventilated and it is safe to start fighting fire at site B, etc. However, agents can communicate through the environment, e.g., if agent FB2 successfully ventilates site A it can infer without asking agents FB1 and FB3 that the fire at site A has been put out.

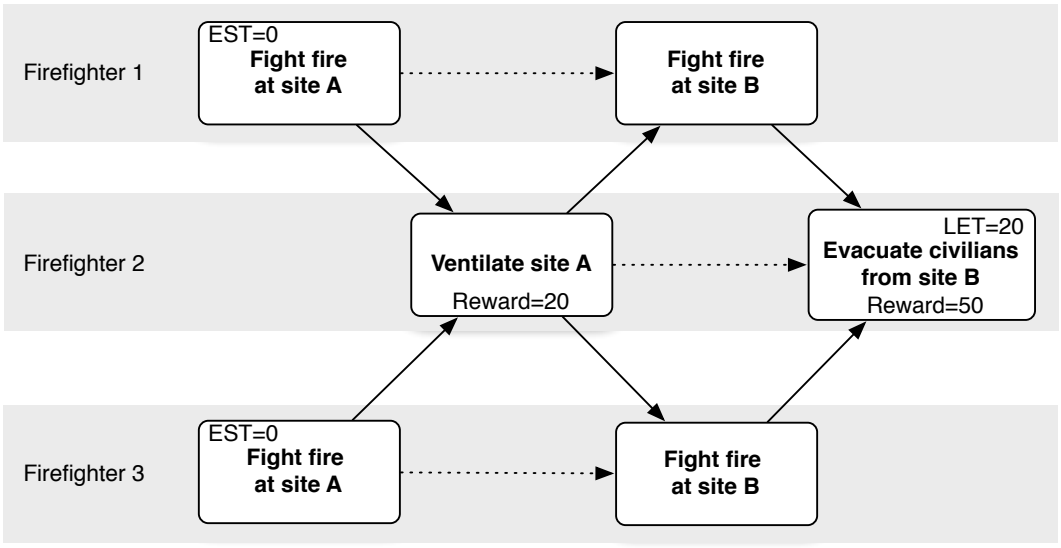

Figure 2.3: Civilian rescue domain and a mission plan

One can clearly see the dilemma that FF2 faces: It can only estimate the durations of the "Fight fire at site A" methods executed by FF1 and FF3 and at the same time, FF2 knows that time is running out for civilians. If FF2 ventilates site A too early (before the fire at site A is put out by agents FF1 and FF3), the fire will spread out of control. On the other hand, if FF2 waits

with the ventilation of site A for too long, fire at site B will become unbearable for the civilians. In general, agents have to perform a sequence of such difficult decisions. For example, a decision process of FF2 involves first choosing when to start ventilating site A, and then (depending on the time it took to ventilate site A), choosing when to start evacuating the civilians from site B. Such sequence of decisions constitutes the policy of an agent, and it must be found fast because time is running out.

2.1.3.2 Example 2: Partially Ordered Domain

This section illustrates a multiagent domain where the activities of the agents are only partially ordered. Consider a planning problem in Figure 2.4 where a team of agents has to perform a set of activities in a continuous time interval $[\tau_s, \tau_e]$. The agent team consists of two agents, Agent 1 and Agent 2 on a mission to perform methods m_1, m_2, m_3 and m_4, m_5, m_6 respectively *in any order*. Furthermore, (i) each agent can be executing only one method at a time, (ii) execution of methods m_4, m_2 must be preceded by the successful completion of methods m_1 and m_5 respectively and (iii) method execution durations are sampled from a normal distribution with a mean $\mu = 4$ and variance $\sigma = 2$. The goal of each agent is to maximize the team reward earnings for rewards $r_1 = 1$, $r_2 = 5$, $r_3 = 2$, $r_4 = 5$, $r_5 = 1$, $r_6 = 2$ earned upon the successful execution of methods $m_1, m_2, m_3, m_4, m_5, m_6$ respectively.

Consider the situation where, at mission start time τ_s, Agent 1 and Agent 2 are choosing which methods should they start executing first. Since r_2 and r_4 are much greater than rewards for other methods, Agent 1 might be tempted to first execute method m_1 in order to allow Agent 2 to successfully start the execution of its method m_4 early enough, so that the execution of

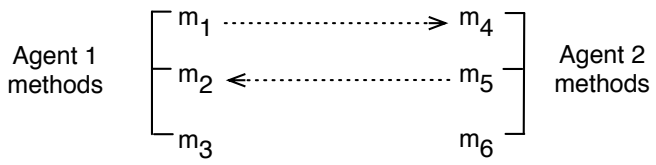

Figure 2.4: Partially ordered multiagent domain

method m_4 can finish before the mission deadline. On the other hand, Agent 2 might be tempted to execute method m_5 first such that Agent 1 has enough time to start and finish the execution of its method m_2 before the mission deadline. Clearly, this inconsistency has to be resolved as there might be not enough time (before the mission deadline τ_e) to execute both, the execution sequence m_1 followed by m_4 and the execution sequence m_5 followed by m_2.

Suppose that Agent 1 decides that, together with Agent 2, they will attempt to execute the sequence m_1 followed by m_4. The plan is that Agent 1 will start executing m_1 while Agent 2 will wait until time t_1 after which it will start executing its method m_4. When the execution of m_4 ends, e.g. at time t_2, Agent 2 will make a decision whether to start executing m_5 or m_6. If there is enough time left, i.e., if $\tau_e - t_2$ is sufficiently big, Agent 2 will consider that there is still enough time for the sequence of methods m_5 followed by m_2 to be executed successfully. Otherwise, it will simply decide to go after method m_6, which is worth more than method m_5 alone. Note a complication here: t_2 is not known before the mission starts and thus, to evaluate the utility of the sequence m_1 followed by m_4, one would have to consider an infinite number of values t_2 which is impossible.

Notice how the environment is used for communication in this domain: If agent 2 executes method m_4 within its time window but this execution is unsuccessful, agent 2 can immediately infer that agent 1 has not executed its method m_1 successfully. Similarly, if agent 1 executes

method m_2 within its time window and this execution is successful, agent 1 can immediately infer that agent 2 has successfully executed method m_5 before agent 1 has started executing method m_2. Thus, similarly to [Becker et al., 2004] special purpose methods can be added to play a role in exchanging the information between the agents.

2.1.4 General Description of Multiagent Domains

This section provides a general description of multiagent planning problems analyzed in this thesis. It assumes assumes a team of agents deployed on a mission to perform a given set of tasks, referred to as methods. Such formulation has received a lot of attention in the literature. In particular, it is inspired by the DARPA Coordinators effort Musliner et al. [2006], which in turn is based on the popular GPGP paradigm Decker and Lesser [1995].

The formal description of the multiagent planning problems considered in this thesis is as follows: A team of N agents is deployed on a mission to perform a set of methods from the set $M = \{m_1, ..., m_K\}$. Each agent n is assigned to a set M_n of methods, such that $\{M_n\}_{n=1}^{N}$ is a partitioning of $\{m_k\}_{k=1}^{K}$. Furthermore, each agent can be executing only one method at a time, each method can be executed only once and each method execution consumes a certain amount of agent resource — it is assumed that there is only one type of resource. Also, the probability density function p_k of the amount of resource consumed by the execution of method m_k is assumed to be known, for all $k = 1, .., K$. For example, every agent knows that method m_k will consume t amount of resource with probability $p_k(t)$.

Furthermore, the planning problems of interests are specified by two types of constraints: *Resource precedence* and *resource limit* constraints with the following meaning:

- Resource precedence constraints grouped in set $C_{\prec}$ impose restrictions on method execution based on resource levels of other methods. Precisely, a resource precedence constraint $\langle i, j \rangle \in C_{\prec}$ is a binary relation that links methods $m_i, m_j \in M$ that belong to possibly different agents. The constraint imposes two necessary (but not sufficient) conditions on the successful execution of method m_j. These are: (i) Method m_i must be executed successfully and (ii) if the execution of method m_i finishes with l amount of resource left, the execution of method m_j must start with *less* than l resource left. For example, if time-to-deadline is a resource, condition (ii) states that method m_j must be started *after* method m_i is finished.

  It is often referred to that method m_j is **enabled** at resource level l if all the methods m_i such that $\langle i, j \rangle \in C_{\prec}$ have been successfully finished with more than l resources left. For example, if time is a resource than method m_j is enabled at time t if all the methods m_i such that $\langle i, j \rangle \in C_{\prec}$ have been successfully finished before time t. To illustrate this concept consider Figure 2.3. Here, method "Ventilate site A" is enabled by two methods: "Fight fire at site A" of Firefighter 1 and "Fight fire at site A" of Firefighter 3.

  Checking whether for a given joint policy a resource precedence constraint is met is trivial if methods m_i and m_j belong to the same agent. However, it is more problematic if methods m_i and m_j belong to different agents because it is assumed that agents cannot directly communicate. Indeed, it is by checking whether the method m_j has been executed successfully / unsuccessfully that allows an agent to learn whether method m_i has been executed successfully / unsuccessfully. Hence, checking the satisfiability of resource precedence constraints *post factum* plays a key role in inter-agent communication (see also Section 2.1.3.2).

- Resource limit constraints grouped in set $C_{[]}$ impose restrictions on internal agent resource levels during method execution. Precisely, a resource limit constraint $\langle i, l_1, l_2 \rangle \in C_{[]}$ for a method $m_i \in M$ imposes that the resource levels of an agent executing method m_i must stay within the range $[l_1, l_2]$. For example, if absolute time is a resource, method m_i should be started *after* time l_1 and finished *before* time l_2. It follows from the resource limit constraints in $C_{[]}$ that the maximum resource level to be considered when constructing an optimal plan is $\Delta = \max\{l_2 : \langle i, l_1, l_2 \rangle \in C_{[]}\}$. Finally, if absolute time is a resource, Δ is referred to as the mission deadline.

The goal of the agents is to maximize the joint reward earned by the agent team. The individual reward r_i for executing method m_i is earned when: (i) Resource limit constraints for method m_i are met and (ii) method m_i is enabled when its execution starts. If condition (i) is met but condition (ii) is not met, the execution of method m_i proceeds normally (as if m_i was enabled when it was started), but the execution of m_i is unsuccessful and reward r_i is not earned. Also, when during the execution of method m_i condition (i) stops to hold (agent resource level drops too low) the execution of method m_i is interrupted, method m_i is considered to be executed unsuccessfully and reward r_i is not earned.

2.2 Formal Models

For modeling the planning problems such as the ones introduced in Section 2.1, in this section I introduce two formal frameworks: (i) Continuous Resource MDP — for single agent planning problems and (ii) Continuous Resource, Decentralized MDP — for multiagent planning problems. To this end, I first provide the formal definition of the Markov Decision Process.

2.2.1 Markov Decision Process

Markov Decision Process (MDP) [Puterman, 1994] is a mathematical model for decision making where the outcomes of actions are stochastic. MDP assumes a that there is a synchronous dialog between some environment and some decision making entity. At each step of this dialog, the entity makes a decision which affects the environment and then receives an observation which uniquely determines the new state of the environment 2.5. From now on, this entity will be referred to as an *agent*.

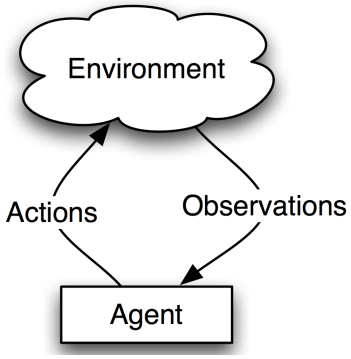

Figure 2.5: Agent interacting with the environment in a Markov Decision Process

Formally MDP is a tuple $\langle S, A, P, R \rangle$ where:

- S is the set of states of the environment;
- A is the set of agent actions and $A(s) \subset A$ is the set of actions that can be executed from state $s \in S$.
- $P : S \times A \times S \mapsto [0, 1]$ is the transition function. For example, $P(s, a, s')$ is the probability that the agent will transition to state $s' \in S$ if it executes action $a \in A(s)$. For all states $s \in S$ and actions $a \in A(s)$ it holds that $\sum_{s' \in S} P(s, a, s') = 1$.

- $R : S \times A \times S \mapsto \mathbb{R}$ is the reward function. For example, $R(s, a, s')$ is the reward that the agent receives when it transitions to state $s' \in S$ after executing an action $a \in A(s)$.

A deterministic, stationary policy π for an MDP $\langle S, A, P, R \rangle$ is a mapping $\pi : S \mapsto A$. For example, $\pi(s) = a$ postulates that an action $a \in A$ will be executed from state $s \in S$. For a policy π and some state $s \in S$, the *total expected utility* $V^{\pi}(s)$ of policy π started in state s can be derived using the following recursive equation:

$$V^{\pi}(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot (R(s, \pi(s), s') + V^{\pi}(s')) \tag{2.1}$$

And the policy π^* is said to be *optimal* if for a given starting state $s_0 \in S$ it holds that $V^{\pi^*}(s_0) \geq V^{\pi}(s_0)$ for any policy $\pi \neq \pi^*$.

2.2.2 Continuous Resource MDP

In order to model decision processes that require reasoning with continuous resources, the standard MDP model has to be extended. In particular, the extended model must capture that: (i) resources are continuous, i.e., resource levels take real values, (ii) resources are always depleting, (iii) resource consumption is stochastic and (iii) resources have both lower and upper limits. This extended model is referred to as a **continuous resource MDP**. First, the most general version of the continuous resource MDP model is defined. Next, a narrowed down version is outlined, to model the planning problems of interest from Sections 2.1.1 and 2.1.2.

Formally, a continuous resource MDP is a tuple $\langle S \times X, A, P, D, R \rangle$, where:

- S is a set of discrete states and $X = \times_{1 \leq i \leq k} X_i$ is a k-dimensional continuous variable whose values identify current resource levels. For example, the MDP can occupy discrete state

$s \in S$ with current resource levels $(x_i)_{1 \leq i \leq k} \in X$ where x_i is the current level of resource i. Furthermore, each continuous resource has lower and upper limits, i.e., $X_i = [a_i, b_i] \subset \mathbb{R}$. *For example, in the planetary exploration domain from Section 2.1.1,* $S = \{start, site1, site2, site3, base\}$ *and* $X = [0, \Delta]$ *with time-to-deadline being a continuous resource.*

- A is a set of actions and $A(s) \subset A$ is the set of actions that can be executed from the discrete state $s \in S$. *For example, in the planetary exploration domain from Section 2.1.1,* $A =$ {*move-to-next-site, return-to-base* }.

- $P : S \times A \times S \mapsto [0, 1]$ is the discrete state transition function. For example, $P(s, a, s')$ is the probability that the agent will transition to discrete state $s' \in S$ if it executes action $a \in A(s)$. For all discrete states $s \in S$ and actions $a \in A(s)$ it holds that $\sum_{s' \in S} P(s, a, s') = 1$. *In the planetary exploration domain from Section 2.1.1, all discrete transitions are deterministic, e.g.* $P(site1, move\text{-}to\text{-}next\text{-}site, site2) = 1$.

- $D : S \times A \mapsto (X_i \mapsto \mathbb{R})_{1 \leq i \leq k}$ is the table of resource consumption distributions. For example, for each action $a \in A(s)$ executed from discrete state $s \in S$ there exists a set $D(s, a) = (p^i_{s,a})_{1 \leq i \leq k}$ of resource consumption probability distribution functions $p^i_{s,a}$ — one distribution function per each resource i. In particular, $p^i_{s,a}(y)$ is the probability that action $a \in A(s)$ executed from discrete state s will consume y amount of resource i. *In the planetary exploration domain from Section 2.1.1, for each state and action the resource consumption distribution is the same, e.g.* $D(site1, move\text{-}to\text{-}next\text{-}site) = f$ *where* $f(t) = e^{-t}$.

- $R : S \times A \times S \times X \mapsto \mathbb{R}$ is the reward function. For example, $R(s, a, s', x)$ is the reward that the agent receives when it executes action $a \in A(s)$ from discrete state $s \in S$ and transitions to a discrete state $s' \in S$ with $x \in X$ amounts of resources left. *In particular, for the*

planetary exploration domain from Section 2.1.1, the rover receives reward $R(site1, return\text{-}to\text{-}base, base, t) = 6$ *provided that* $t > 0$.

For modeling the planning problems from Sections 2.1.1 and 2.1.2, the most general version of the continuous resource MDPs model (defined above) is narrowed down so that it models one continuous resource, namely, *time-to-deadline*. For notational convenience, each discrete state $s \in S$ will be referred to as just *state* whereas current resource level will be referred to as *current time-to-deadline*.

For such MDPs, $X = [0, \Delta]$, where Δ is the maximum time to deadline at the beginning of execution. The agent transitions are the rewards it earns are then the following: Assume that an agent is in discrete state $s \in S$ with a deadline being $t > 0$ time units away (= with time-to-deadline t), after which execution stops. It executes an action $a \in A(s)$ of its choice. The execution of the action cannot be interrupted, and the action duration t' is distributed according to a given probability distribution $p_{s,a}(t')$ that depends on both the state s and the action a. If $t' \geq t$, then the time-to-deadline $t - t' \leq 0$ after the action execution is non-positive, which means that the deadline is reached (resource is depleted) and execution stops. Otherwise, with probability $P(s, a, s')$, the agent obtains reward $R(s, a, s') \geq 0$ and transitions to state $s' \in S$ with time-to-deadline $t - t'$ and repeats the process.

The objective of the agent is to maximize its expected total reward until execution stops. Precisely, Let $V^*(s)(t)$ denote the largest expected total reward that the agent can obtain until

execution stops if it starts in state $s \in S$ with time-to-deadline $0 \leq t \leq \Delta$. The agent can maximize its expected total reward by executing the action

$$\pi^*(s)(t) \arg\max_{a \in A(s)} \left\{ \sum_{s' \in S} P(s, a, s') \int_0^t p_{s,a}(t')(R(s, a, s') + V^*(s')(t - t'))\, dt' \right\}$$

in state $s \in S$ with time-to-deadline $0 \leq t \leq \Delta$, which can be explained as follows: When it executes action $a \in A(s)$ in state $s \in S$, it incurs action duration t' with probability $p_{s,a}(t')$. If $0 \leq t' \leq t$, then it transitions to state $s' \in S$ with probability $P(s, a, s')$ and obtains an expected total future reward of $R(s, a, s') + V^*(s')(t - t')$. The function π^* is the optimal agent policy that needs to be found.

As will be discussed in Chapter 7, there are serious shortcomings in previous work in determining π^*. In essence, current algorithms either (i) make additional restrictive assumptions about the model or (ii) do not add new assumptions, but run very slow (see Section 1.2). This thesis addresses both of these shortcomings in two contributions:

- It introduces CPH, a fast analytic algorithm for solving continuous resource MDPs. CPH solves the planning problems at hand by first approximating with a desired accuracy the probability distributions over the resource consumptions with phase-type distributions, which use exponential distributions as building blocks. It then uses value iteration to solve the resulting MDPs up to three orders of magnitude faster than its closest competitor, and allows for a systematic trade-off of solution quality for speed.

- Second, to improve the anytime performance of CPH and other continuous resource MDP solvers this thesis introduces the DPFP algorithm. Rather than using the principle of value iteration to solve the planning problems at hand, DPFP performs a forward search in the

corresponding dual space of cumulative distribution functions. In doing so, DPFP discriminates in its policy generation effort providing only approximate policies for regions of the state-space reachable with low probability yet it bounds the error that such approximation entails.

2.2.3 Continuous Resource, Decentralized MDP

This section introduces a formal framework for planning problems outlined in Section 2.1.4. To this end, the concepts of execution events and histories of execution events must be defined:

Definition 1. *An* **execution event** *stores all the relevant information about the execution of a method. Precisely, an execution event e is a tuple* $\langle i, l_1, l_2, q \rangle$ *where:*

- i *is the index of method* $m_i \in M_n$ *executed by some agent n;*
- l_1 *is agent n resource level when it started executing method* m_i*;*
- l_2 *is agent n resource level when it finished executing method* m_i*;*
- $q \in \{0, 1\}$ *is the result of the execution of method* m_i*. If* $q = 1$ *method* m_i *has been executed successfully, whereas if* $q = 0$ *method* m_i *has been executed unsuccessfully.*

Furthermore, a ***spoof*** *execution event* $\langle 0, l_1, l_2, 1 \rangle$ *is used to specify an event when the agent remained idle (executed a spoof method* m_0*) and its resource level dropped from* l_1 *to* l_2*.*

Definition 2. *An* **execution history** *stores the complete history of execution events of an agent. Precisely, the execution history h of an agent n is a vector* $(e_1, e_2, ..., e_k)$ *where* $e_1, e_2, ..., e_k$ *are the execution events of agent n.*

Observe, that agent n knows exactly its execution history. However, because it is assumed that agents cannot communicate directly their execution status, agent n might not know exactly the execution histories of other agents. Indeed, agent n only knows the probability distribution of the execution histories of other agents. This distribution is affected by both: (i) The joint policy of all the agents that has been found during the planning phase and (i) the observations that agent n received during the execution phase, where an observation is an outcome of the execution of a method involved in a resource precedence constraint (explained in Section 5.2.2).

The formal definition of the Continuous Resource, Decentralized MDP (CR-DEC-MDP) model is then the following:

Definition 3. *For a team of N agents, a CR-DEC-MDP is a collection of N continuous resource Markov decision processes* $\mathcal{MDP}_n = \langle \mathcal{S}_n, \mathcal{A}_n, \mathcal{P}_n, \mathcal{R}_n \rangle$ *where:*

- $\mathcal{S}_n$ is the set of states of agent n where each state $s \in S_n$ is the agent execution history, e.g. $s = (e_1, ..., e_k)$. Each $\mathcal{MDP}_n$ has a distinguished starting state $s_{n,0} \in \mathcal{S}_n$ encoded as $s_{n,0} = (\langle -n, l_{n,0}, l_{n,0}, 1 \rangle) \in S_n$ where m_{-n} is a unique spoof method of agent n which by default is completed in the starting state with $l_{n,0}$ resource left — the only purpose of the starting state is to encode the initial resource level of the agent. The current state $s \in S_n$ of agent n allows the agent to estimate the probability distribution over current states of MDPs of other agents (explained in Section 5.2.2).

- $\mathcal{A}_n$ is the set of actions of agent n. Furthermore, $A(s) \subset \mathcal{A}_n$ is used to denote a set of actions that agent n can execute in state $s \in S_n$. Because methods are allowed to be executed only once, $A(s) = M_n \setminus \{m_i : \langle i, l_1, l_2, q \rangle \in s\}$. Alternatively, an agent can choose to remain idle in which case its resource level drops by the controllable amount δ.

- $\mathcal{P}_n$ is the state transition function of agent n that can depend on the current state of $\mathcal{MDP}_{n'}$ of any agent n'. When the agent is in state $s = (\langle i_1, l_{1,1}, l_{1,2}, q_1\rangle, ..., \langle i_k, l_{k,1}, l_{k,2}, q_k\rangle)$ it can either choose to remain idle (deliberately lose a certain amount of resource) or execute a method $m_j \in A(s)$. If the agent chooses to remain idle, it looses a controllable amount δ of resource and transitions to a state $s' = (\langle i_1, l_{1,1}, l_{1,2}, q_1\rangle, ..., \langle i_k, l_{k,1}, l_{k,2}, q_k\rangle, \langle 0, l_{k,2}, l_{k,2} - \delta, 1\rangle)$. Else, if the agent decides to execute a method $m_j \in A(s)$, it transitions with probability $p_j(x)$ to state $s' = (\langle i_1, l_{1,1}, l_{1,2}, q_1\rangle, ..., \langle i_k, l_{k,1}, l_{k,2}, q_k\rangle, \langle j, l_{k,2}, l_{k,2} - x, q_j\rangle)$ where x is the amount of resource consumed during the execution of method m_j. The result q_j of the execution of method m_j is 1 (method m_j has been executed successfully) if and only if both the resource limit constraint and resource precedence constraint are satisfied. Precisely:

  - The resource limit constraint is met when the agent resource level remains within some admissible range during the execution of method m_j, i.e., if there exist a resource limit constraint $\langle j, l_1, l_2\rangle \in C_{[]}$ such that $l_1 \geq l_{k,2} \geq l_{k,2} - x \geq l_2$.
  - The resource precedence constraint is met if method m_j is enabled when it is started, i.e., if the execution of all methods $m_i \in M$ such that $\langle i, j\rangle \in C_{\prec}$ has finished successfully with at least l_{k_2} resources left. In other words, for each method $m_i \in M_{n'}$ such that $\langle i, j\rangle \in C_{\prec}$, the current state of $\mathcal{MDP}_{n'}$ must contain an event $e = \langle i, l_1, l_2, 1\rangle$ where $l_2 \geq l_{k,2}$.

  Finally, since it is assumed that the execution of method m_j is automatically interrupted when the agent resource level drops below value l_2, for some admissible resource range constraint $\langle j, l_1, l_2\rangle \in C_{[]}$ the transition function must be modified respectively. To this end,

the probability that the execution of method m_j consumes $l_{k,2} - l_2$ amount of resource must be $1 - \int_0^{l_{k,2}-l_2} p_j(x)dx$.

- $\mathcal{R}_n$ is the reward function of agent n. If the agent executes a method $m_j \in A(s)$ from state $s = (e_1, ..., e_k)$ and transitions to state $s' = (e_1, ..., e_k, \langle j, l_{k,1}, l_{k_2}, q_j \rangle)$ it earns reward $q_j \cdot r_j$.

Given local policies π_n of agents $n = 1, .., N$ the joint policy is a vector $(\pi_1, ..., \pi_n)$. The value of the joint policy is then defined as follows:

Definition 4. *The value of the joint policy* $\pi = (\pi_1, ..., \pi_N)$ *is given by* $V(\pi) = \sum_{n=1}^{N} V_{\pi_i}(s_{n,0})$ *where* $V_{\pi_i}(s_{n,0})$ *is the total expected reward of policy* π_n *(of agent n) executed from its starting state* $s_{n,0}$. *The* **optimal policy** $\pi^* = (\pi_1^*, ..., \pi_N^*)$ *is the one that maximizes* $V(\pi)$.

As will be discussed in Chapter 7, there are serious shortcomings in previous work in determining π^*. First, current algorithms are only applicable to CR-DEC-MDPs where the resource levels are discretized and second, current algorithms may still exhibit poor performance with the scale-up in their state-spaces. To remedy these shortcomings, this thesis proposes two different algorithms for solving CR-DEC-MDPs (see Chapter 5):

- The first algorithm (VFP) considers a special case when CR-DEC-MDPs are fully ordered (explained in Section 5.1). For such CR-DEC-MDPs, VFP finds locally optimal policies one order of magnitude faster than its competitors. In addition, VFP implements a set of heuristics aimed at improving the quality of these locally optimal joint policies.

- The second proposed algorithm (M-DPFP) for solving CR-DEC-MDPs operates on arbitrary CR-DEC-MDPs. It finds joint policies that are guaranteed to be within an arbitrary small ϵ from the optimal joint policies.

Both VFP and M-DPFP allow for continuous resource levels and continuous resource consumption probability density functions.

Chapter 3: Single Agent Solutions

Two algorithms for solving planning problems modeled as continuous resource MDPs are proposed in this chapter: (i) The value iteration algorithm CPH and (ii) the forward search approach DPFP. The advantages and disadvantages of CPH and DPFP are demonstrated in Chapter 4.

3.1 Value Iteration Approach: The CPH Algorithm

Recall (see Equation 2.2) that the optimal policy π^* is calculated from values $V^*(s)(t) \;\; s \in S; t \in [0, \Delta]$ where $V^*(s)(t)$ is the largest expected total reward that the agent can obtain until execution stops if it starts in state $s \in S$ with time-to-deadline $0 \leq t \leq \Delta$. To calculate values $V^*(s)(t) \;\; s \in S; t \in [0, \Delta]$ necessary to determine π^*, value iteration first calculates the values $V^n(s)(t)$ using the following Bellman updates for all states $s \in S$, time-to-deadlines $0 \leq t \leq \Delta$, and iterations $n \geq 0$:

$$V^0(s)(t) := 0$$

$$V^{n+1}(s)(t) := \begin{cases} 0 & \text{if } t \leq 0 \\ \max_{a \in A(s)} \{\sum_{s' \in S} P(s,a,s') & \text{otherwise} \\ \qquad \int_0^t p_{s,a}(t')(R(s,a,s') + V^n(s')(t-t'))\, dt'\}. & \end{cases}$$

It then holds that $\lim_{n\to\infty} V^n(s)(t) = V^*(s)(t)$ for all states $s \in S$ and times-to-deadline $0 \leq t \leq \Delta$.

Unfortunately, value iteration cannot be implemented as stated since the number of values $V^n(s)(t)$ is infinite for each iteration n since the time-to-deadline t is a real-valued variable. This thesis remedies this situation by viewing the $V^n(s)$ as value functions that map times-to-deadline t to the corresponding values $V^n(s)(t)$. As a result CPH can maintain $V^n(s)$ with a finite number or parametrized, continuous functions. In this context, I make the following contributions:

- First, I show that once the starting MDP is approximated with an MDP with only exponential action duration distributions, it is possible to find such n, that running the value iteration for n iterations would produce a solution with an approximation error $\max_{s\in S, 0\leq t\leq\Delta} |V^*(s)(t) - V^n(s)(t)|$ no larger than a given constant $\epsilon > 0$.

- Second, I show that each value function $V^n(s)$ can be represented exactly with a vector of a small number of real values each.

- Finally, I show how the Bellman updates can efficiently transform the vectors of the value functions $V^n(s)$ to the vectors of the value functions $V^{n+1}(s)$.

These three contributions are presented in the following three steps of the CPH algorithm (**C**ontinuous resource MDP through **PH**ase-type approximation).

3.1.1 CPH Step 1: Phase Type Approximation

CPH first approximates the probability distributions $p_{s,a}$ of action durations that are not exponential with phase-type distributions (see Appendix A), resulting in chains of exponential distributions $E(\lambda) = \lambda e^{-\lambda t}$ with potentially different exit rate parameters $\lambda > 0$ [Neuts, 1981]. CPH then uses uniformization [Puterman, 1994] to make all constants λ identical without changing the underlying stochastic process. The detailed description of these two steps is provided in Appendix C and Appendix D — this section only shows an example of use of phase-type approximation and uniformization in context of CPH. Furthermore, this section reports on the implications of phase-type approximation on the CPH's planning horizon.

The actual implementation of CPH used for the experimental evaluation of CPH (see Chapter 4) employed the Coxian family of distributions (see Appendix B) for phase-type approximation. The uniformization process (see Appendix D) was then used to uniformize the exit rate parameters of the exponential distributions of the underlying Coxian distributions. The example of this process is shown in Figure 3.1. Here, an action duration $p_{s,a}$ follows a Normal distribution with a mean $\mu = 2$ and standard deviation $\sigma = 1$. This normal distribution is first approximated with a 3-phase Coxian distribution and then uniformized.

Let a =*return-to-base*, $s = start$, $s' = base$ and $P(s, a, s') = 1$ as shown in Figure 3.1. First, the Expectation-Maximization algorithm [Dempster et al., 1977] is used to approximate the normal distribution with a 3-phase Coxian distribution. The phase type approximation introduces 2

auxiliary states Ph_2 and Ph_3 (also referred to as phases). Two other phases of the phase-type distribution, Ph_1 and Ph_4, are the original states of a continuous resource MDP: Ph_1 corresponds to state *start* and Ph_4 (the absorbing phase) corresponds to state *base*. Observe, that all phase transition duration times follow exponential distributions — in particular, for the Coxian distribution shown, the exit rates of these exponential distribution and the discrete phase-to-phase transition probabilities are shown in Figure 3.1.

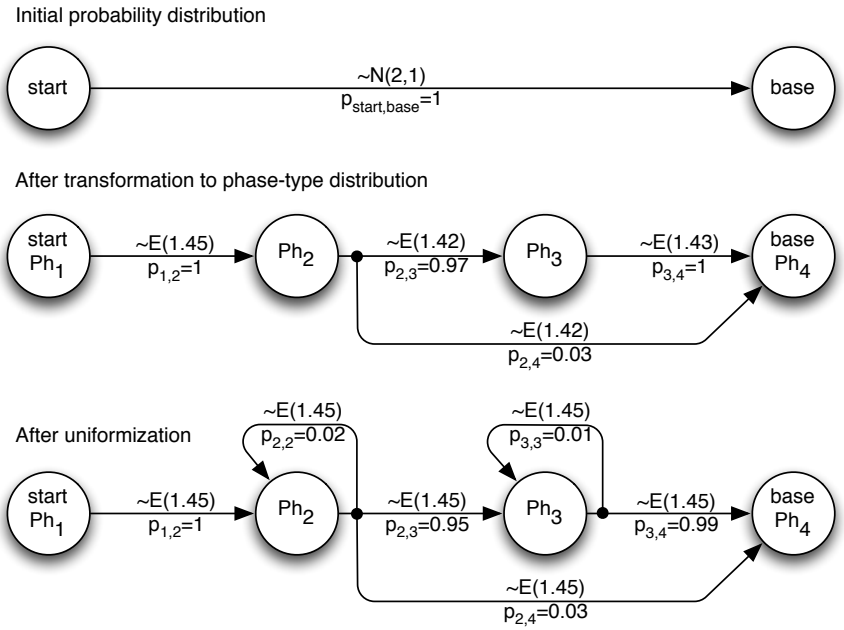

Figure 3.1: Phase type approximation example

A phase type distribution f specifies the probabilities of transitioning to the absorbing phase (to state *base* in the example above) over time. Formally, a phase-type distribution f is given by:[1]

$$f(\vec{\alpha}, Q)(t) = -\vec{\alpha} e^{Qt}(-Q\vec{1}) \tag{3.1}$$

[1]For a detailed description of phase-type distributions, refer to Appendix A

where Q is the infinitesimal generator matrix, $\overrightarrow{\alpha}$ is a vector on starting distribution over Markov states and $\overrightarrow{1}$ is the unit column vector. In particular, for the example shown, $\overrightarrow{\alpha} = (1, 0, 0)$ (the Markov process starts in phase Ph_1) and Q is given by[2]:

$$Q = \begin{pmatrix} -1.45 & 1 \cdot 1.45 & 0 \\ 0 & -1.42 & 0.97 \cdot 1.42 \\ 0 & 0 & -1.43 \end{pmatrix} \tag{3.2}$$

A common measure of fitness of the phase-type approximation is the KL-divergence [Kullback and Leibler, 1951]. In particular, the KL-divergence between the original function (normal distribution with mean $\mu = 2$ and variance $\sigma = 1$) and the obtained phase-type distribution is -1.370521. For details on how to measure the KL-divergence, refer to Section C.

To allow CPH to compute the underlying convolution operations analytically (see Equation 3.1), all phase-type distributions must consists of the exponential distributions that have the same exit rate parameter λ. In order to achieve that, each phase-type distribution is uniformized (for details on the uniformization process, refer to Section D). The uniformization does not alter the generator matrix Q and as such, it does not affect the absorption time of the phase-type distribution. However, uniformization changes the exit rates of the phase-to-phase transition durations as well as the phase-to-phase transition probabilities (intuitively, the exit rates are increased but so are the probabilities of self-transitions). For the example above, the new exit rate ($\lambda = 1.45$) and the new phase-to-phase transition probabilities are shown in Figure 3.1.

From now on, CPH can therefore assume without loss of generality that the action durations t' of all actions are distributed according to exponential distributions $p_{s,a}(t') = p(t') \sim E(\lambda)$

[2]For details on how to determine the values of $\overrightarrow{\alpha}$ and Q refer to Appendix A and Appendix B.

with the same constant $\lambda > 0$. Note however, that the uniformization process can introduce self-transitions, and consequently, CPH's value iteration cannot determine the value functions $V^*(s)$ with a finite number of iterations. To remedy that, Theorem 3.13 from Section 3.1.4 proves that the approximation error is no larger than a desired $\epsilon > 0$ if the value iteration runs for at least n^* iterations where:

$$n^* \geq \log_{\frac{e^{\lambda\Delta}-1}{e^{\lambda\Delta}}} \left(\frac{\epsilon}{R_{max}(e^{\lambda\Delta} - 1)} \right) \tag{3.3}$$

and $R_{max} := \max_{s \in S, a \in A(s), s' \in S} R(s, a, s')$.

3.1.2 CPH Step 2: Piecewise Gamma Value Functions

In this section it is explained how CPH breaks up each value function $V^n(s)(t)$ into multiple sub-functions $V_i^n(s)(t)$ where each sub-function is associated with a particular time interval. Informally, for different time intervals (separated by special times-to-deadline referred to as breakpoints), the value function $V^n(s)(t)$ will be characterized by a different parametric function. Furthermore, each parametric function will have an associated action that an agent should start executing if the current time-to-deadline is inside the time interval of that parametric function.

Formally, it follows directly from the considerations in the next section that there exist times-to-deadline $0 = t_{s,0} < t_{s,1} < \ldots < t_{s,m_s+1} = \Delta$ such that the value function $V^n(s)$ is made of $m_s + 1$ parametric functions $V_i^n(s)$ such that $V^n(s)(t) = V_i^n(s)(t)$ for all $t \in [t_{s,i}, t_{s,i+1})$, where

$$V_i^n(s)(t) = c_{s,i,1} - e^{-\lambda t} \left(c_{s,i,2} + \ldots + c_{s,i,n+1} \frac{(\lambda t)^{n-1}}{(n-1)!} \right) \tag{3.4}$$

for the parameters $c_{s,i,j}$ for all $s \in S$, $0 \leq i \leq m_s$ and $1 \leq j \leq n+1$. These times-to-deadline $t_{s,i}$ are referred to as *breakpoints*, $[t_{s,i}, t_{s,i+1})$ as *intervals*, the expressions for the parametric functions $V_i^n(s)$ as *gamma functions* (which is a simplification since each function $V_i^n(s)$ is actually linear combinations of Euler's incomplete gamma functions), and the expressions for the value functions $V^n(s)$ as piecewise gamma functions. Each gamma function $V_i^n(s)$ is represented as a vector $[c_{s,i,1}, \ldots, c_{s,i,n+1}] = [c_{s,i,j}]_{j=1}^{n+1}$ and each piecewise gamma function $V^n(s)$ as a vector of vectors $[t_{s,i}{:}V_i^n(s)]_{i=0}^{m_s} = [t_{s,i}{:}[c_{s,i,j}]_{j=1}^{n+1}]_{i=0}^{m_s}$.

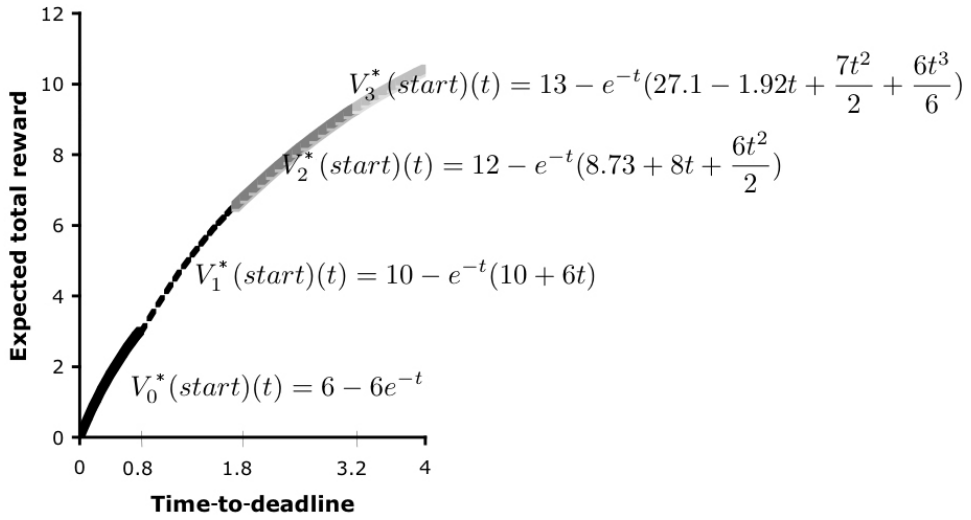

Figure 3.2: CPH Value function $V^*(start)$ consists of four gamma functions: $V^*(start) = [0{:}V_0^*(start),\ 0.8{:}V_1^*(start),\ 1.8{:}V_2^*(start),\ 3.2{:}V_3^*(start)] = [0{:}[6,6],\ 0.8{:}[10,10,6],\ 1.8{:}[12,8.73,8,6],\ 3.2{:}[13,27.1,-1.92,7,6]]$.

Figure 3.2 illustrates the different concepts introduced in this section. The Figure shows the optimal value function $V_0^*(start)$ for state *start* of the planetary exploration domain introduced in Section 2.1.1. Observe that $V_0^*(start)$ is represented by four parametric gamma functions $V_0^*(start)$, $V_1^*(start)$, $V_2^*(start)$, $V_3^*(start)$ associated with four different time intervals $[0, 0.8]$,

[0.8, 1.8], [1.8, 3.2], [3.2, 4.0] respectively — times-to-deadline 0.8, 1.8, 3.2 are the breakpoints of the value function $V_0^*(start)$. Although not shown explicitly in Figure 3.2, the actions associated with the gamma functions are:

- *Return-to-base* if time-to-deadline is between 0 and 0.8 (for function $V_0^*(start)$);
- *Move-to-next-site* if time-to-deadline is between 0.8 and 1.8 (for function $V_1^*(start)$);
- *Move-to-next-site* if time-to-deadline is between 1.8 and 3.2 (for function $V_2^*(start)$);
- *Move-to-next-site* if time-to-deadline is between 3.2 and 4 (for function $V_3^*(start)$);

3.1.3 CPH Step 3: Functional Value Iteration

3.1.3.1 Simplified Example

It is first shown how value iteration can calculate the vectors of the value functions for a *simplified example* where an agent executes an unconditional sequence of deterministic actions $a_1, a_2, \ldots, a_{N+1}$ in its initial state s_1 with time-to-deadline Δ and then execution stops. Thus, a_2 is executed after a_1, a_3 is executed after s_2, ..., a_{N+1} is executed after a_N. Furthermore, assume that the execution of action a_n in state s_n results (with probability one) in state s_{n+1} for all $1 \leq n \leq N$. Then, the Bellman updates that are important (necessary to derive the value functions) are:

$$\begin{aligned} V^0(s_{N+1})(t) &= 0 \\ V^n(s_{N+1-n})(t) &= \int_0^t p(t') \left(R(s_{N+1-n}, a_{N+1-n}, s_{N+2-n}) + V^{n-1}(s_{N+2-n})(t-t') \right) dt' \end{aligned}$$

for all $0 \leq t \leq \Delta$ and $1 \leq n \leq N$. The integral in the second equation is an example of the convolution operation $\int_0^t p(t') f(t-t')\, dt'$ commonly denoted as $(p * f)(t)$. This convolution operation

is now used recursively to transform the Bellman updates to the vector notation introduced in Section 3.1.2:

$$
\begin{aligned}
V^0(s_{N+1}) &= 0 \\
V^n(s_{N+1-n}) &= p * (R(s_{N+1-n}, a_{N+1-n}, s_{N+2-n}) + V^{n-1}(s_{N+2-n})) \\
&= p * R(s_{N+1-n}, a_{N+1-n}, s_{N+2-n}) + p * V^{n-1}(s_{N+2-n}) \\
&\quad \text{(since convolution is distributive)} \\
&= p * R(s_{N+1-n}, a_{N+1-n}, s_{N+2-n}) \\
&\quad + p * p * R(s_{N+2-n}, a_{N+2-n}, s_{N+3-n}) \\
&\quad + \dots \\
&\quad + \underbrace{p * \dots * p}_{n-1} * R(s_{N-1}, a_{N-1}, s_N) \\
&\quad + \underbrace{p * \dots * p}_{n} * R(s_N, a_N, s_{N+1}) \\
&\quad \text{(the recursion has been "unrolled")}
\end{aligned}
$$

Now, for $p(t)$ sampled from an exponential distribution $E(\lambda) = \lambda e^{-\lambda t}$ and a constant R it can easily be derived that $\underbrace{p * \dots * p}_{n} * R = R - e^{-\lambda t}\left(R + \dots + R\frac{(\lambda t)^{n-1}}{(n-1)!}\right)$. Furthermore, this expression

can be stored as a vector $[\underbrace{R, ..., R}_{n+1}] = [R]_{i=1}^{n+1}$ in accordance with the vector notation introduced in Section 3.1.2, Hence, the derivation of $V^n(s_{N+1-n})$ continues:

$$\begin{aligned} = \quad & [R(s_{N+1-n}, a_{N+1-n}, s_{N+2-n})]_{j=1}^{2} \\ & + [R(s_{N+2-n}, a_{N+2-n}, s_{N+3-n})]_{j=1}^{3} \\ & + \ldots \\ & + [R(s_{N-1}, a_{N-1}, s_N)]_{j=1}^{n} \\ & + [R(s_N, a_N, s_{N+1})]_{j=1}^{n+1} \end{aligned}$$

Finally, the above vectors (of different sizes) are summed into a vector $[c_1, c_2, ..., c_{N+1}]$ using the following method: c_n is the sum of the n-th elements in the above vectors, for vectors that have at least n elements. Thus:

$$V^n(s_{N+1-n}) = [\sum_{j=N+1-n}^{N} R(s_j, a_j, s_{j+1}), \sum_{j=N+1-n}^{N} R(s_j, a_j, s_{j+1}), \sum_{j=N+2-n}^{N} R(s_j, a_j, s_{j+1}), \ldots, \sum_{j=N}^{N} R(s_j, a_j, s_{j+1})]$$

for all $1 \leq n \leq N$. Having transformed the Bellman updates to vector notation, this notation for $n = N$ can be used in expressing the value function for state s_1:

$$V^N(s_1) \quad = \quad [\sum_{j=1}^{N} R(s_j, a_j, s_{j+1}), \sum_{j=1}^{N} R(s_j, a_j, s_{j+1}), \sum_{j=2}^{N} R(s_j, a_j, s_{j+1}), \ldots, \sum_{j=N}^{N} R(s_j, a_j, s_{j+1})].$$

Example: For the planetary exploration domain, recall the optimal value function $V^*(start)$ from Figure 3.2. According to this function, the optimal action for the time interval $[0, 0.8]$

is to return-to-base and the expected utility of this action over time is given by $V_0^*(start)(t) = 6 \cdot (1 - e^{-\lambda t})$[3]. *This result is now verified; when the rover executes the "return-to-base" action unconditionally at state "start" and then execution stops, it receives reward 6 for completing the action. Thus, from Equation 3.5,* $V^1(start)(t) = \lambda e^{\lambda t} * 6 = 6 \cdot (1 - e^{-\lambda t}) = [6, 6]$[4]. *Similarly, when the rover executes the "return-to-base" action unconditionally at state "start" and then execution stops, it holds that* $V^1(site_3)(t) = \lambda e^{\lambda t} * 6 = 6 \cdot (1 - e^{-\lambda t}) = [6, 6]$.

3.1.3.2 The General Case

In the previous section, only for demonstration purposes, it has been assumed that an agent executes an unconditional sequence of deterministic actions. Solving the planning problems, however, is more complicated since it might not be optimal to, say, always execute action a_2 after action a_1 no matter how long it takes to execute action a_1 or which state results from its execution. This is the reason for why for different time intervals, the value function is represented by a different gamma function (instead of being represented by a single gamma function for all $0 \leq t \leq \Delta$).

It is now shown how to generalize the key idea to allow value iteration to solve the planning problems analytically. For $n = 0$, the value functions $V^n(s) = 0$ satisfy $V^n(s) = [0{:}[0]]$ and thus are (piecewise) gamma. It is shown by induction that all value functions $V^{n+1}(s)$ are piecewise gamma if all value functions $V^n(s)$ are piecewise gamma. It is also shown how the Bellman updates can efficiently transform the vectors of the value functions $V^n(s)$ to the vectors of the value functions $V^{n+1}(s)$. Recall Equation 3.1 which performs the value iteration:

[3]The lower index in V_0^* represents an index of the gamma function

[4]The upper index in V^1 represents the Bellman update number, not the gamma function index. Indeed, the value functions for an unconditional sequence of actions considered in this section are always represented by a single gamma function

$$V^{n+1}(s)(t) \quad := \quad \max_{a \in A(s)} \sum_{s' \in S} P(s,a,s') \int_0^t p(t')(R(s,a,s') + V^n(s')(t-t'))\, dt'.$$

This calculation breaks down into four stages:

- First, $\overline{V}^n(s')(t-t') := R(s,a,s') + V^n(s')(t-t')$.
- Second, $\widetilde{V}^n(s')(t) := \int_0^t p(t')\overline{V}^n(s')(t-t')\, dt'$.
- Third, $\widehat{V}^n(s,a)(t) := \sum_{s' \in S} P(s,a,s')\widetilde{V}^n(s')(t)$.
- Finally, $V^{n+1}(s)(t) := \max_{a \in A(s)} \widehat{V}^n(s,a)(t)$.[5]

Stage 1: Calculate $\overline{V}^n(s')(t-t') := R(s,a,s') + V^n(s')(t-t')$. The induction assumption is that all value functions $V^n(s')$ are piecewise gamma, i.e., $V^n(s') = [t_{s',i}{:}[c_{s',i,j}]_{j=1}^{n+1}]_{i=0}^{m_{s'}}$. In Stage 1, CPH calculates $\overline{V}^n(s')(t-t') := R(s,a,s') + V^n(s')(t-t')$, which is the same as calculating $\overline{V}^n(s')(t) := R(s,a,s') + V^n(s')(t)$ since $R(s,a,s')$ is constant. Then,

$$\begin{aligned} \overline{V}^n(s') &= R(s,a,s') + V^n(s') \\ &= R(s,a,s') + [t_{s',i} : [c_{s',i,j}]_{j=1}^{n+1}]_{i=0}^{m_{s'}} \\ &= [t_{s',i} : [c'_{s',i,j}]_{j=1}^{n+1}]_{i=0}^{m_{s'}}. \end{aligned}$$

where $c'_{s',i,1} = R(s,a,s') + c_{s',i,1}$ and $c'_{s',i,j} = c_{s',i,j}$ for all $0 \le i \le m_{s'}$ and $2 \le j \le n+1$.

Example (continued): In the previous example for the planetary exploration domain it has been found, that $V^1(site_3) = [6,6]$. *This example is now carried on in order to show how CPH*

[5] One should really use $\overline{V}^n(s,a,s')(t-t')$ and $\widetilde{V}^n(s,a,s')(t)$ instead of $\overline{V}^n(s')(t-t')$ and $\widetilde{V}^n(s')(t)$, respectively, but this would make the notation rather cumbersome.

derives $V^2(site_2)$*. If in state* $site_2$ *the rover executes the "move to site 3" action, then the value function after Step 1 is* $\overline{V}^1(site_3) = [1+6, 6] = [7, 6]$ *because* $V^1(site_3) = [6, 6]$.

Stage 2: Calculate $\widetilde{V}^n(s')(t) := \int_0^t p(t')\overline{V}^n(s')(t-t')\,dt'$, which is a convolution of p and $\overline{V}^n(s')$ denoted as $p * \overline{V}^n(s')$. Consider the value functions $\overline{V}^n(s')$ defined in Stage 1. It is now shown by induction that

$$\begin{aligned}\widetilde{V}_i^n(s')(t) &= (p * \overline{V}_i^n(s'))(t) - e^{-\lambda t}\Big(\sum_{i'=1}^{i} e^{\lambda t_{s',i'}}(p * \overline{V}_{i'}^n(s'))(t_{s',i'})\\ &- \sum_{i'=0}^{i-1} e^{\lambda t_{s',i'+1}}(p * \overline{V}_{i'}^n(s'))(t_{s',i'+1})\Big) \qquad (3.5)\end{aligned}$$

for all $t \in [t_{s',i}, t_{s',i+1})$. Equation (3.5) holds for $i = 0$ since $\widetilde{V}^n(s')(t) = (p * \overline{V}_0^n(s'))(t)$ for all $t \in [t_{s',0}, t_{s',1})$. Assume now that Equation (3.5) holds for some i. It then also holds for $i+1$ as shown in Figure 3.3. It has consequently been shown that the transformation performed at stage 2 results in a value function $\widetilde{V}^n(s')(t)$ that is piecewise, and that each piece $\widetilde{V}_i^n(s')(t)$ is a function represented by Equation (3.5). In order to complete stage 2, it must also be shown, that $\widetilde{V}^n(s')(t)$ is in piecewise gamma form i.e., it must be shown that each piece $\widetilde{V}_i^n(s')$ is a gamma function:

Lemma 1. *Let* $p(t)$ *follow the exponential probability density function with the exit rate* λ*, i.e.,* $p(t) \sim E(\lambda)$*. Then,* $p * [k_1, k_2, \ldots, k_n] = [k_1, k_1, k_2, \ldots, k_n]$.

Proof. By symbolic integration, $p * [k]_{j=1}^n = [k]_{j=1}^{n+1}$ for any constant k. Then, $p * [k_1, \ldots, k_n] = p*(\sum_{i=1}^{n-1}[k_i - k_{i+1}]_{j=1}^i + [k_n]_{j=1}^n) = \sum_{i=1}^{n-1}(p*[k_i - k_{i+1}]_{j=1}^i) + p*[k_n]_{j=1}^n = \sum_{i=1}^{n-1}[k_i - k_{i+1}]_{j=1}^{i+1} + [k_n]_{j=1}^{n+1} = [k_1, k_1, \ldots, k_n]$. □

The recursion will be proven for $i+1$. Since convolution is commutative it holds that:

$$\widetilde{V}^n_{i+1}(s')(t) = \int_0^t p(t')\overline{V}^n(s')(t-t')\,dt' = \int_0^t p(t-t')\overline{V}^n(s')(t')\,dt'$$

The integral range $[0,t]$ is now split into ranges $[0,t_{s',i+1}]$ and $(t_{s',i+1},t]$:

$$= \int_{t_{s',i+1}}^t p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' + \int_0^{t_{s',i+1}} p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt'$$

For all t' such that $t_{s',i+1} \le t' \le t < t_{s',i+2}$ it holds that $\overline{V}^n(s')(t') = \overline{V}^n_{i+1}(s')(t')$. Thus:

$$= \int_{t_{s',i+1}}^t p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' + \int_0^{t_{s',i+1}} p(t-t')\overline{V}^n(s')(t')\,dt'$$

By adding and then subtracting $[0,t_{s',i+1}]$ to the first integral range:

$$= \int_0^t p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' - \int_0^{t_{s',i+1}} p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' + \int_0^{t_{s',i+1}} p(t-t')\overline{V}^n(s')(t')\,dt'$$

And since $p(t-t') = \lambda e^{-\lambda(t-t')} = e^{-\lambda(t-t_{s',i+1})}\lambda e^{-\lambda(t_{s',i+1}-t')} = e^{-\lambda(t-t_{s',i+1})}p(t_{s',i+1}-t')$ it holds that:

$$= \int_0^t p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' - e^{-\lambda(t-t_{s',i+1})}\int_0^{t_{s',i+1}} p(t_{s',i+1}-t')\overline{V}^n_{i+1}(s')(t')\,dt' + e^{-\lambda(t-t_{s',i+1})}\int_0^{t_{s',i+1}} p(t_{s',i+1}-t')\overline{V}^n(s')(t')\,dt'$$

Now, by the definition of $\widetilde{V}^n_i(s')$:

$$= \int_0^t p(t-t')\overline{V}^n_{i+1}(s')(t')\,dt' - e^{-\lambda(t-t_{s',i+1})}\int_0^{t_{s',i+1}} p(t_{s',i+1}-t')\overline{V}^n_{i+1}(s')(t')\,dt' + e^{-\lambda(t-t_{s',i+1})}\widetilde{V}^n_i(s')(t_{s',i+1})$$

And compact notation for the convolution $p * \overline{V}^n_{i+1}(s')$:

$$= (p * \overline{V}^n_{i+1}(s'))(t) - e^{-\lambda(t-t_{s',i+1})}(p * \overline{V}^n_{i+1}(s'))(t_{s',i+1}) + e^{-\lambda(t-t_{s',i+1})}\widetilde{V}^n_i(s')(t_{s',i+1})$$

Finally, $\widetilde{V}^n_i(s')(t)$ is unrolled by using the induction assumption (for an argument $t = t_{s',i+1}$):

$$\begin{aligned} &= (p * \overline{V}^n_{i+1}(s'))(t) - e^{-\lambda(t-t_{s',i+1})}(p * \overline{V}^n_{i+1}(s'))(t_{s',i+1}) \\ &\quad + e^{-\lambda(t-t_{s',i+1})}\Big((p * \overline{V}^n_i(s'))(t_{s',i+1}) \\ &\quad - e^{-\lambda t_{s',i+1}}\left(\sum_{i'=1}^{i} e^{\lambda t_{s',i'}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'}) - \sum_{i'=0}^{i-1} e^{\lambda t_{s',i'+1}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'+1})\right)\Big) \end{aligned}$$

And since all the terms except $(p * \overline{V}^n_{i+1}(s'))(t)$ are multiplications of $e^{\lambda t}$ it holds that:

$$\begin{aligned} &= (p * \overline{V}^n_{i+1}(s'))(t) - e^{-\lambda t}e^{\lambda t_{s',i+1}}(p * \overline{V}^n_{i+1}(s'))(t_{s',i+1}) + e^{-\lambda t}e^{\lambda t_{s',i+1}}(p * \overline{V}^n_i(s'))(t_{s',i+1}) \\ &\quad - e^{-\lambda t}\left(\sum_{i'=1}^{i} e^{\lambda t_{s',i'}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'}) - \sum_{i'=0}^{i-1} e^{\lambda t_{s',i'+1}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'+1})\right) \\ &= (p * \overline{V}^n_{i+1}(s'))(t) - e^{-\lambda t}\left(\sum_{i'=1}^{i+1} e^{\lambda t_{s',i'}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'}) - \sum_{i'=0}^{i} e^{\lambda t_{s',i'+1}}(p * \overline{V}^n_{i'}(s'))(t_{s',i'+1})\right) \end{aligned}$$

Figure 3.3: Proof by induction of Equation (3.5).

Using the result from the lemma, Equation (3.5) can now be transformed to vector notation.

$$
\begin{aligned}
\widetilde{V}_i^n(s') \quad &= \quad [c''_{s',i,j}]_{j=1}^{n+2} \\
\text{where} \qquad & c''_{s',i,1} := c'_{s',i,1} \qquad c''_{s',i,2} := c'_{s',i,1} + z_{s',i} \\
& c''_{s',i,j+1} := c'_{s',i,j} \quad \forall_{j=2,3,\ldots,n+1} \\
\text{with} \qquad & z_{s',i} := \sum_{i'=1}^{i} e^{\lambda t_{s',i'}} (p * \overline{V}_{i'}^n(s'))(t_{s',i'}) \\
& \qquad - \sum_{i'=0}^{i-1} e^{\lambda t_{s',i'+1}} (p * \overline{V}_{i'}^n(s'))(t_{s',i'+1}).
\end{aligned}
$$

Observe that $z_{s',i}$ is added to $c''_{s',i,2}$ because $z_{s',i}$ is multiplied by $e^{-\lambda t}$ in Equation (3.5). Consequently, $\widetilde{V}^n(s')$ is in piecewise gamma form: $\widetilde{V}^n(s') = [t_{s',i}{:}[c''_{s',i,j}]_{j=1}^{n+2}]_{i=0}^{m_{s'}}$. That result completes all the necessary calculations performed at stage 2.

Note that one could also calculate $\widetilde{V}_{i+1}^n(s')$ recursively based on $\widetilde{V}_i^n(s')$ according to line 5 in Figure 3.3. It then holds that:

$$
\begin{aligned}
\widetilde{V}_{i+1}^n(s')(t) \quad &= (p * \overline{V}_{i+1}^n(s'))(t) - e^{-\lambda(t-t_{s',i+1})}(p * \overline{V}_{i+1}^n(s'))(t_{s',i+1}) \\
&\quad + e^{-\lambda(t-t_{s',i+1})}\widetilde{V}_i^n(s')(t_{s',i+1}) \\
&= (p * \overline{V}_{i+1}^n(s'))(t) - e^{-\lambda t} e^{\lambda t_{s',i+1}}((p * \overline{V}_{i+1}^n(s'))(t_{s',i+1}) \\
&\quad - \widetilde{V}_i^n(s')(t_{s',i+1}))
\end{aligned}
$$

and in vector notation:

$$
\widetilde{V}_{i+1}^n(s') = [c'_{s',i+1,1}, c'_{s',i+1,1} + z'_{s',i+1}, c'_{s',i+1,2}, \ldots, c'_{s',i+1,n+1}]
$$

for

$$z'_{s',i+1} := e^{\lambda t_{s',i+1}}((p * \overline{V}^n_{i+1}(s'))(t_{s',i+1}) - \widetilde{V}^n_i(s')(t_{s',i+1})).$$

Example (continued): In the previous example it has been found that if in state site$_2$ the rover executes the "move to site 3" action, then the value function after Step 1 is $\overline{V}^1(site_3) = [7, 6]$. *Since* $\overline{V}^1(site_3) \equiv \overline{V}^1_1(site_3)$ *(because* $\overline{V}^1(site_3)$ *has no breakpoints), it can be easily computed that after stage 2,* $\widetilde{V}^1(site_3) = p * \overline{V}^1(site_3) = p * [7, 6] = [7, 7, 6]$.

Stage 3: Calculate $\widehat{V}^n(s, a)(t) := \sum_{s' \in S} P(s, a, s')\widetilde{V}^n(s')(t)$. Consider the value functions $\widetilde{V}^n(s')$ defined in Stage 2. Since they might have different breakpoints $\{t_{s',i}\}_{i=0}^{m_{s'}}$ for different states $s' \in S$ with $P(s, a, s') > 0$, CPH introduces the common breakpoints

$$\{t_{s,a,i}\}_{i=0}^{m_{s,a}} = \bigcup_{s' \in S : P(s,a,s') > 0} \{t_{s',i}\}_{i=0}^{m'_s}$$

without changing the value functions $\widetilde{V}^n(s')$. Afterwards, $\widetilde{V}^n(s') = [t_{s,a,i} : [c'''_{s',i,j}]_{j=1}^{n+2}]_{i=0}^{m_{s,a}}$ where $c'''_{s',i,j} = c''_{s',i',j}$ for the unique i' with $[t_{s,a,i}, t_{s,a,i+1}) \subseteq [t_{s',i'}, t_{s',i'+1})$. Then,

$$\begin{aligned}
\widehat{V}^n(s, a) &= \sum_{s' \in S} P(s, a, s')\widetilde{V}^n(s') \\
&= \sum_{s' \in S} P(s, a, s')[t_{s,a,i} : [c'''_{s',i,j}]_{j=1}^{n+2}]_{i=0}^{m_{s,a}} \\
&= [t_{s,a,i} : [\sum_{s' \in S} P(s, a, s')c'''_{s',i,j}]_{j=1}^{n+2}]_{i=0}^{m_{s,a}} \\
&= [t_{s,a,i} : [c''''_{s,a,i,j}]_{j=1}^{n+2}]_{i=0}^{m_{s,a}}.
\end{aligned}$$

Example (continued): In the previous example it has been found that if in state $site_2$ the rover executes the "move to site 3" action, then the value function after Step 2 is $\widetilde{V}^1(site_3) = [7, 7, 6]$. Since the "move to site 3" action is deterministic, i.e., $P(site_2, move\text{-}to\text{-}site\text{-}3, site_3) = 1$, after stage 3 it holds that $\widehat{V}^1(site_2, move\text{-}to\text{-}site\text{-}3) = 1 \cdot [7, 7, 6] = [7, 6, 6]$.

Stage 4: Calculate $V^{n+1}(s)(t) := \max_{a \in A(s)} \widehat{V}^n(s, a)(t)$. After stage 3, for each action $a \in A(s)$ one has a value function $\widehat{V}^n(s, a)$ that has a set $\{t_{s,a,i}\}_{i=0}^{m_{s,a}}$ of breakpoints associated with it. Since for different actions $a \in A(s)$, sets $\{t_{s,a,i}\}_{i=0}^{m_{s,a}}$ can be different, the first thing CPH does in stage 4 is to introduce common breakpoints $\bigcup_{a \in A(s)} \{t_{s,a,i}\}_{i=0}^{m_{s,a}}$ without changing the value functions $\widehat{V}^n(s, a)$. CPH then introduces additional *dominance* breakpoints at the intersections of value functions $\widehat{V}^n(s, a)\ \ \forall_{a \in A(s)}$ to ensure that, over each interval, one of the value functions dominates the other ones. Let $\{t'_{s,i}\}_{i=0}^{m'_s}$ be the set of breakpoints afterwards. Then, $V^{n+1}(s) = [t'_{s,i}{:}[c''''''_{s,i,j}]_{j=1}^{n+2}]_{i=0}^{m'_s}$ where $c''''''_{s,i,j} = c''''_{s,a_{s,i},i',j}$ for actions $a_{s,i} \in A(s)$ and the unique i' with $[t'_{s,i}, t'_{s,i+1}) \subseteq [t_{s,a_{s,i},i'}, t_{s,a_{s,i},i'+1})$ and, for all $t \in [t'_{s,i}, t'_{s,i+1})$, $\widehat{V}^n(s, a)(t) \leq \widehat{V}^n(s, a_{s,i})(t)$. Furthermore, action $a_{s,i}$ should be executed according to the value function $V^{n+1}(s)$ if the current state is s and the time-to-deadline is $t \in [t'_{s,i}, t'_{s,i+1})$.

Example (continued) : It has already been shown that when the rover executes the "move to site 3" action, after stage 3 it holds that $\widehat{V}^1(site_2, move\text{-}to\text{-}site\text{-}3) = [7, 6, 6]$. One can calculate in the similar fashion, that when the rover executes the "move to base" action from $site_2$, after stage 3: $\widehat{V}^1(site_2, move\text{-}to\text{-}base) = [6, 6, 0]$. Stage 4 calculates the maximum of $\widehat{V}^1(site_2, move\text{-}to\text{-}site\text{-}3)$ and $\widehat{V}^1(site_2, move\text{-}to\text{-}base)$. The two functions intersect at 3.06 and the maximum is $V^2(site_2) = [0 : [6, 6, 0], 3.06 : [7, 7, 6]$. Now, to show how to calculate $V^3(site_1)$: If the rover

executes the "move to base" action, then (as before) after stage 1 one obtains: $[6+0,0,0] = [6,0,0]$ *and after stage 2 one obtains:* $[6,6,0,0]$. *If the rover executes the "move to site 2" action, then one obtains after stage 1:* $[0:[2+6,6,0],3.06:[2+7,7,6] = [0:[8,6,0],3.06:[9,7,6]$ *and after stage 2:* $[0:[8,8,6,0],3.06:[9,-2.09,7,6]]$. *Stage 4 calculates the maximum of* $[6,6,0,0]$ *and* $[0:[8,8,6,0],3.06:[9,-2.09,7,6]]$. *The two functions intersect at 1.87 and the maximum is* $V^3(site_1) = [0:[6,6,0,0],1.87:[8,8,6,0],3.06:[9,-2.09,7,6]]$.

To summarize, the value functions $V^{n+1}(s)$ are piecewise gamma, and the vectors of the value functions $V^n(s)$ can be transformed automatically to the vectors of the value functions $V^{n+1}(s)$. The lengths of the vectors increase linearly in the number of iterations and, although the number of breakpoints (that are placed automatically during the transformations) can increase exponentially, in practice it stays small since one can merge small intervals after each iteration of value iteration to reduce the number of breakpoints The transformations require only simple vector manipulations and a numerical method that determines the dominance breakpoints approximately, for which CPH uses a bisection method. In Chapter 4 it is shown experimentally that these transformations of CPH are both efficient and accurate.

3.1.4 Error Control

The following theorem assumes that all action durations follow phase-type distributions. It does not take into account the error introduced by approximating non-phase type distributions with phase-type distributions:

Theorem 1. *Let $\epsilon > 0$ be any positive constant, $R_{max} := \max_{s \in S, a \in A(s), s' \in S} R(s, a, s')$ and*

$$n \geq \log_{\frac{e^{\lambda\Delta}-1}{e^{\lambda\Delta}}} \left(\frac{\epsilon}{R_{max}(e^{\lambda\Delta} - 1)} \right).$$

It then holds that:

$$\max_{s \in S, 0 \leq t \leq \Delta} |V^*(s)(t) - V^n(s)(t)| \leq \epsilon.$$

Proof. Let $\alpha := \lambda\Delta > 0$ and $b_i := \sum_{j=i}^{\infty} \frac{\alpha^j}{j!}$ for all $i \geq 0$. It holds that $b_0 = e^\alpha$ and $b_1 = b_0 - 1 = e^\alpha - 1$. First, a bound on the probability $p_i(t)$ that the sum of the action durations of a sequence of $i \geq 1$ actions is no more than $0 \leq t \leq \Delta$ is provided:

$$\begin{aligned} p_i(t) &\leq p_i(\Delta) = \int_0^\Delta (\underbrace{p * p * \ldots * p}_{i})(t')\, dt' = \int_0^\Delta e^{-\lambda t'} \frac{t'^{i-1}\lambda^i}{i!}\, dt' \\ &= \frac{1}{e^\alpha} \left(e^\alpha - \sum_{j=0}^{i-1} \frac{\alpha^j}{j!} \right) = \frac{1}{e^\alpha} \left(\sum_{j=0}^{\infty} \frac{\alpha^j}{j!} - \sum_{j=0}^{i-1} \frac{\alpha^j}{j!} \right) \\ &= \frac{1}{e^\alpha} \sum_{j=i}^{\infty} \frac{\alpha^j}{j!} = \frac{b_i}{e^\alpha}. \end{aligned}$$

The values

$$\frac{b_{i+1}}{b_i} = \frac{b_{i+1}}{\frac{\alpha^i}{i!} + b_{i+1}} = \frac{1}{\frac{\alpha^i}{i!\, b_{i+1}} + 1}$$

decrease strictly monotonically in i because the values

$$\frac{\alpha^i}{i!\, b_{i+1}} = \frac{\alpha^i}{i!\, \sum_{j=i+1}^{\infty} \frac{\alpha^j}{j!}} = \frac{1}{\frac{\alpha}{i+1} + \frac{\alpha^2}{(i+2)(i+1)} + \frac{\alpha^3}{(i+3)(i+2)(i+1)} + \ldots}$$

increase strictly monotonically in i. Consequently,

$$1 > \frac{e^{\alpha} - 1}{e^{\alpha}} = \frac{b_1}{b_0} > \frac{b_2}{b_1} > \frac{b_3}{b_2} > \ldots > 0.$$

Thus

$$\begin{aligned} b_i &< \frac{b_{i-1}}{b_{i-2}} b_{i-1} < \frac{b_1}{b_0} b_{i-1} < \frac{b_1}{b_0} \frac{b_{i-2}}{b_{i-3}} b_{i-2} < \left(\frac{b_1}{b_0}\right)^2 b_{i-2} \\ &< \ldots < \left(\frac{b_1}{b_0}\right)^{i-1} b_1. \end{aligned}$$

These results are now used to bound $|V^*(s)(t) - V^n(s)(t)|$ for all $s \in S$ and $0 \leq t \leq \Delta$. Assume that the agent starts in state $s \in S$ with time-to-deadline $0 \leq t \leq \Delta$. Value iteration with n iterations determines the highest expected total reward $V^n(s)(t)$ under the restriction that execution stops when the deadline is reached or n actions have been executed. The largest expected total reward $V^*(s)(t)$ does not have the second restriction and can thus be larger than $V^n(s)(t)$. In particular, only $V^*(s)(t)$ takes into account that the agent can execute the $(n+1)$st action with probability $p_{n+1}(t)$, the $(n+2)$nd action with probability $p_{n+2}(t)$, and so on, receiving a reward of

at most R_{max} for each action execution. Additionally, $V^n(s)(t)$ is locally greedy, i.e., the reward for its first n actions exceeds the reward for the first n actions of $V^*(s)(t)$. Thus,

$$\begin{aligned}
0 &\leq V^*(s)(t) - V^n(s)(t) \leq \sum_{i=n+1}^{\infty} R_{max} p_i(t) \\
&\leq \frac{R_{max}}{e^{\alpha}} \sum_{i=n+1}^{\infty} b_i < \frac{R_{max}}{e^{\alpha}} \sum_{i=n+1}^{\infty} \left(\frac{b_1}{b_0}\right)^{i-1} b_1 \\
&= \frac{R_{max} b_1}{e^{\alpha}} \sum_{i=0}^{\infty} \left(\frac{b_1}{b_0}\right)^{i+n} = \frac{R_{max} b_1}{e^{\alpha}} \left(\frac{b_1}{b_0}\right)^{n} \sum_{i=0}^{\infty} \left(\frac{b_1}{b_0}\right)^{i} \\
&= \frac{R_{max} b_1}{e^{\alpha}} \left(\frac{b_1}{b_0}\right)^{n} \frac{1}{1 - \frac{b_1}{b_0}}.
\end{aligned}$$

The goal of this theorem is to bound this expression by ϵ. If value functions have breakpoints, chosen error ϵ must be first reduced by the error ξ introduced by breakpoints (ξ is small and can be easily bounded by $\kappa^{n^*}\mu$ where μ is the bisection method error, κ is the maximum number of breakpoints and n is the planning horizon). Let $\epsilon' := \epsilon - \xi > 0$, then:

$$\begin{aligned}
\frac{R_{max} b_1}{e^{\alpha}} \left(\frac{b_1}{b_0}\right)^{n} \frac{1}{1 - \frac{b_1}{b_0}} &\leq \epsilon' \\
\left(\frac{b_1}{b_0}\right)^{n} &\leq \frac{\epsilon'(1 - \frac{b_1}{b_0})e^{\alpha}}{R_{max} b_1} \\
n &\geq \log_{\frac{b_1}{b_0}} \left(\frac{\epsilon'(1 - \frac{b_1}{b_0})e^{\alpha}}{R_{max} b_1}\right) \\
n &\geq \log_{\frac{e^{\lambda\Delta}-1}{e^{\lambda\Delta}}} \left(\frac{\epsilon'}{R_{max}(e^{\lambda\Delta} - 1)}\right).
\end{aligned}$$

□

3.2 Forward Search Approach: The DPFP Algorithm

In this section the **D**ynamic **P**robability **F**unction **P**ropagation (DPFP) approach for solving continuous resource MDPs is introduced. DPFP alleviates the major weakness of CPH, that is, CPHs poor anytime performance (see Chapter 4). DPFP approach is a novel combination of three key ideas:

- It introduces the concept of forward search to a continuous resource MDP setting;
- Its forward search is performed in a dual space of cumulative distribution functions which allows it to vary the planning effort based on the likelihood of reaching different regions of the state-space;
- It bounds the error that such approximations entails.

These three features allow for a superior anytime performance of DPFP when compared to CPH or other algorithms for solving continuous resource MDPs. Furthermore, they allow DPFP to be run in a hybrid mode with CPH or other continuous resource MDP solvers, to speed up the search for high quality solutions.

3.2.1 DPFP at a Glance

Before the DPFP algorithm is described formally, the informal intuition behind this approach is provided. The ability of CPH and other value iteration algorithms to find π^* comes at a high price. Indeed, value iteration propagates values backwards, and thus, in order to find $\pi^*(s_0)(0)$, it must first find $\pi^*(s)(t)$ for all states $s \in S$ reachable from s_0 and all $t \in [0, \Delta]$ — no matter how likely it is that state s is visited at time t (in the Mars rover domain with 10 sites of interest, value iteration

must plan for all 2^{10} states and for all $t \in [0, \Delta]$). In fact, value iteration does not even know the probabilities of transitioning to a state s at time t prior to finding π^*.

DPFP on the other hand, as a forward search algorithm, can determine the probabilities of transitioning to a state s at time t prior to finding π^*. Hence, DPFP can discriminate in its policy generation effort providing only approximate policies for pairs (s, t) encountered with low probability. Unfortunately if an MDP contains cycles or action duration distributions are continuous, standard forward search cannot be carried out in a standard way as it would have to consider an infinite number of candidate policies.

To remedy that, DPFP exploits two insights. First, since each action consumes a certain minimum amount of time, only a finite number of actions can be performed before the deadline Mausam et al. [2005] and thus, the action horizon of DPFP's forward search can be finite. Second, to avoid having to consider an infinite number of policies when action duration distributions are continuous, DPFP operates on a different search space referred to as the dual space of cumulative distribution functions. In that dual space, DPFP only finds approximate solutions, yet it can express the error of its approximations in terms of an arbitrary small parameter κ. This process will now be explained in detail.

3.2.2 Dual Problem

There exists an alternative technique for determining π^* that does not use Equation 2.2, and thus, does not calculate the values $V^*(s)(t)$ for all $s \in S$ and $t \in [0, \Delta]$. For notational convenience, from now on, and until the end of this chapter *absolute-time* (not time-to-deadline) is considered to be a continuous resource. In other words, the process starts at time $t = 0$ and terminates at time $t = \Delta$ and each action increases the current absolute time. Note, that replacing time-to-deadline

with absolute-time does not affect the underlying planning problems, because time-to-deadline is simply Δ minus absolute-time. Consequently, the policy for absolute times maps directly to the policy for times-to-deadline and vice versa. In the following, the absolute-time is simply referred to as time.

Let $\phi = (s_0, ..., s)$ be an execution path that starts in state s_0 at time t_0 and finishes in state s. $\Phi(s_0)$ is a set of all paths reachable from state s_0. Also, let $F^*(\phi)(t)$ be the probability of completing the traversal of path ϕ *before* time t when following the optimal policy π^*, and $F^*(\phi, a)(t)$ be the probability of completing the traversal of path ϕ and starting the execution of action $a \in A(s)$ *before* time t when following policy π^* — both $F^*(\phi)$ and $F^*(\phi, a)$ are cumulative distribution functions over $t \in [0, \Delta]$. In this context, the optimal *deterministic* policy $\pi^*(s)$ for state s can be calculated as follows:

$$\pi^*(s)(t) = \arg\max_{a \in A(s)} \left\{ \lim_{\epsilon \to 0} F^*(\phi, a)(t + \epsilon) - F^*(\phi)(t) \right\} \tag{3.6}$$

Since the derivative of $F^*(\phi, a)$ with respect to time is positive at time t for only one action $a \in A(s)$. The set $F^* := \{F^*(\phi); F^*(\phi, a)$ for all $\phi = (s_0, ..., s) \in \Phi(s_0)$ and $a \in A(s)\}$ is referred to as the solution to the dual problem.

It is now shown how to find F^*. For simplicity (but without the loss of generality) assume that rewards are only dependent on the state that the process transitions to, i.e., $R(s)$ is the reward for executing any action from any state and transitioning to state $s \in S$. Since rewards $R(s)$ are

earned upon entering states $s \in S$ before time Δ, the expected utility $V^{\pi}(s_0)(0)$ of a policy π is given by:

$$V^{\pi}(s_0)(0) = \sum_{\phi=(s_0,...,s)\in\Phi(s_0)} F^{\pi}(\phi)(\Delta) \cdot R(s)$$

Where F^{π} differs from F^{*} in that F^{π} is associated with policy π rather than π^{*}. Since solution F^{*} must yield $V^{*}(s_0)(0)$, it has to satisfy:

$$\begin{aligned} V^{*}(s_0)(0) &= \max_{\pi} V^{\pi}(s_0)(0) \\ &= \max_{\pi} \sum_{\phi=(s_0,...,s)\in\Phi(s_0)} F^{\pi}(\phi)(\Delta) \cdot R(s) \\ &= \sum_{\phi=(s_0,...,s)\in\Phi(s_0)} F^{*}(\phi)(\Delta) \cdot R(s) \end{aligned}$$

In addition, $F^{*} \in X = \{F : (3.7), (3.8), (3.9)\}$ where:

$$F((s_0))(t) = 1 \tag{3.7}$$

$$F((s_0, ..., s))(t) = \sum_{a\in A(s)} F((s_0, ..., s), a)(t) \tag{3.8}$$

$$\begin{aligned} F((s_0, ..., s, s'))(t) &= \sum_{a\in A(s)} P(s, a, s') \\ &\quad \cdot \int_0^t F((s_0, ..., s), a)(t') \cdot p_{s,a}(t - t')dt' \end{aligned} \tag{3.9}$$

In the above set, constraint (3.7) ensures that the process starts in state s_0 at time 0. Constraint (3.8) can be interpreted as the conservation of probability mass flow through path $(s_0, ..., s)$; Applicable only if $|A(s)| > 0$, it ensures that the cumulative distribution function $F((s_0, ..., s))$ is split into cumulative distribution functions $F((s_0, ..., s), a)$ for $a \in A(s)$. Finally, constraint (3.9)

ensures the correct propagation of probability mass $F(s_0, ..., s, s')$ from path $(s_0, ..., s)$ to path $(s_0, ..., s, s')$. It ensures that path $(s_0, ..., s, s')$ is traversed at time t if path $(s_0, ..., s)$ is traversed at time $t' \in [0, t]$ and then, action $a \in A(s)$ takes time $t - t'$ to transition to state s'. The dual problem is then stated as:

$$\max \sum_{\phi=(s_0,...,s)\in\Phi(s_0)} F(\phi)(\Delta) \cdot R(s) \quad | \quad F \in X$$

3.2.3 Solving the Dual Problem

In general, the dual problem is extremely difficult to solve optimally because when action duration distributions are continuous or the MDP has cycles, the set X where F^* is to be found is infinite. Yet, it is now shown that even if action duration distributions are continuous and the MDP has cycles, the dual problem can be solved near-optimally with guarantees on solution quality. The idea of the algorithm that is proposed is to restrict the search for F^* to finite number of elements in X by pruning from X the elements F that correspond to reaching regions of the state-space with very low probability. In essence, when the probability of reaching certain regions of the state-space is below a given threshold, the expected quality loss for executing suboptimal actions in these regions can be bounded, and this quality loss can be traded off for speed.

More specifically, the algorithm searches for F^* in set $\widehat{X} \subset X$ where $\widehat{X}$ differs from X in that values of functions F in $\widehat{X}$ are restricted to integer multiples of a given $\kappa \in \mathbb{R}^+$. Informally, κ

creates a step function approximation of F. Formally, $\widehat{X} = \{F : (3.7), (3.8), (3.10), (3.11), (3.12)\}$ where

$$F'((s_0, ..., s, s'))(t) = \sum_{a \in A(s)} P(s, a, s') \cdot \int_0^t F((s_0, ..., s), a)(t') \cdot p_{s,a}(t - t')dt' \quad (3.10)$$

$$F((s_0, ..., s, s'))(t) = \lfloor F'((s_0, ..., s, s'))(t)/\kappa \rfloor \cdot \kappa \quad (3.11)$$

$$F((s_0, ..., s), a)(t) = \kappa \cdot n \quad \text{where} \quad n \in N \quad (3.12)$$

The *restricted dual problem* is then stated as:

$$\max \sum_{\phi=(s_0,...,s) \in \Phi(s_0)} F(\phi)(\Delta) \cdot R(s) \quad | \quad F \in \widehat{X}$$

Note, that since $\widehat{X}$ is finite, the restricted dual problem can be solved optimally by iterating over all elements of $\widehat{X}$. In the following an algorithm that carries out this iteration is shown; the algorithm returns a policy $\widehat{\pi^*}$ that is guaranteed to be at most ϵ away from π^* where ϵ can be expressed in terms of κ. The algorithm is first shown on an example. Then, the pseudo-code of the algorithm is provided. Finally, the error bound of the algorithm is established.

Figure 3.4 shows the algorithm in action. Assume, $A(s_0) = \{a_1\}; A(s_1) = A(s_2) = \{a_1, a_2\}$; $A(s_3), A(s_4), A(s_5)$ is arbitrary. Also, $P(s_0, a_1, s_1) = P(s_1, a_1, s_2) = P(s_1, a_2, s_3) = P(s_2, a_1, s_4) = P(s_2, a_2, s_5) = 1$ and $\kappa = 0.2$. The algorithm iterates over all elements in $\widehat{X}$. It starts with $F((s_0))$ which is given by constraint (3.7), then uses constraints (3.8), (3.10) to derive $F'((s_0, s_1))$ (solid gray line for state s_1) and finally uses constraint (3.11) to approximate $F'((s_0, s_1))$ with a step function $F((s_0, s_1))$ (solid black line for state s_1).

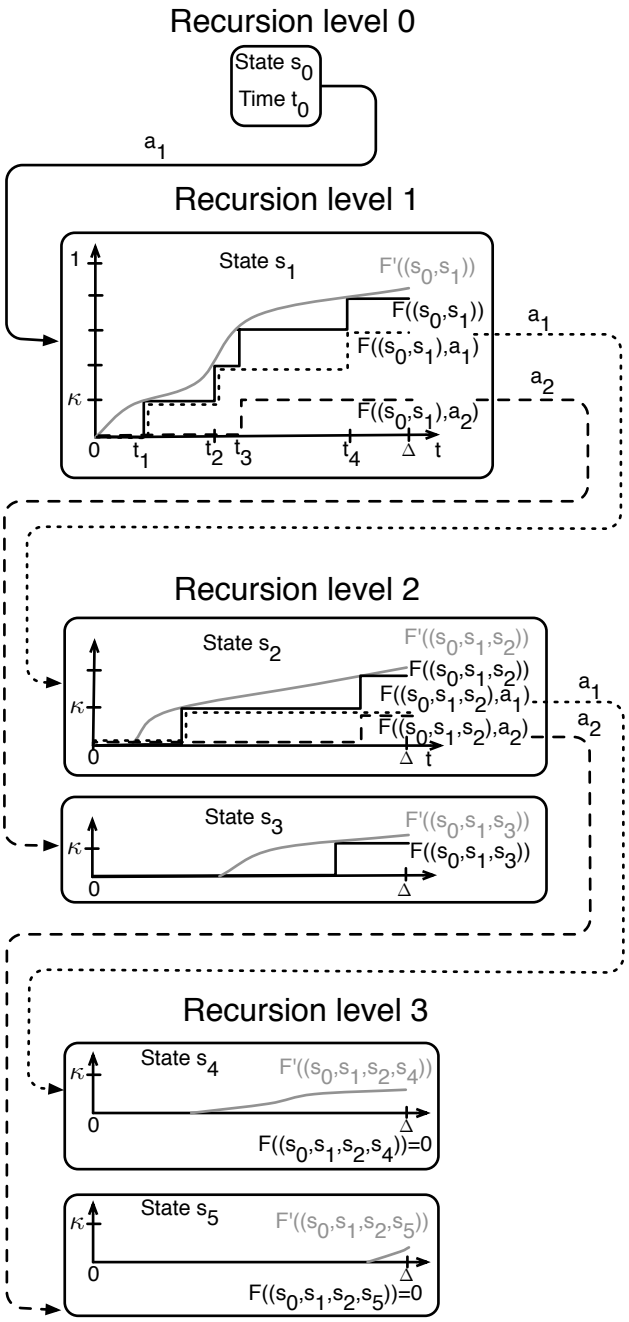

Figure 3.4: Forward search for an optimal probabilistic policy in an approximate space of cumulative distribution functions

At this point the algorithm knows the probability $F((s_0, s_1))(t)$ that s_1 will be visited before time t but does not know the probabilities $F((s_0, s_1), a_1)(t)$ $F((s_0, s_1), a_2)(t)$ that actions a_1 or a_2 will be started from s_1 before time t (dotted black lines for state s_1). Thus, to iterate over all elements in $\widehat{X}$, it must iterate over all $|A(s_1)|^{F((s_0,s_1))(\Delta)/\kappa} = 16$ different sets of functions $\{F((s_0, s_1), a_1); F((s_0, s_1), a_2)\}$ (also called *splittings* of $F((s_0, s_1))$). A splitting determines the policy (see Equation 3.6). In particular, for the specific splitting shown, action a_1 is started at times t_1, t_2, t_4 whereas action a_2 is started at time t_3 (it is shown later show how to extrapolate this policy on $[0, \Delta]$).

At this point, the algorithm calls itself recursively. It now knows $F((s_0, s_1, s_2))$ and $F((s_0, s_1, s_3))$ (derived from $\{F((s_0, s_1), a_1); F((s_0, s_1), a_2)\}$ using constraints (3.10), (3.11)) but does not know the probabilities $F((s_0, s_1, s_2), a_1)(t)$; $F((s_0, s_1, s_2), a_2)(t)$ that actions a_1 or a_2 will be started from s_2 before time t. Thus, to iterate over all elements in $\widehat{X}$, it iterates over all sets $\{F((s_0, s_1, s_2), a_1)$; $F((s_0, s_1, s_2), a_2)\}$ (splittings of $F((s_0, s_1, s_2))$). In this case, for the specific splitting shown, $F((s_0, s_1, s_2, s_4))(\Delta) < \kappa$ and thus, no splittings of $F((s_0, s_1, s_2, s_4))$ are possible (similarly for $F((s_0, s_1, s_2, s_5))$). In such case, the algorithm stops iterating over policies for states following s_4, because the maximum reward loss for not planning for these states, bounded by $\kappa \cdot R$ where $R = \sum_{\phi=(s_0,s_1,s_2,s_4,...,s)} R(s)$, can be made arbitrary small by choosing a sufficiently small κ (to be shown later). Thus, the algorithm evaluates the current splitting of $F((s_0, s_1, s_2))$ and continues iterating over remaining splittings of $F((s_0, s_1, s_2))$ after which it backtracks and picks another splitting of $F((s_0, s_1))$ etc.

In general, the algorithm is started by calling DPFP$((s_0), 1)$ for a globally defined and arbitrary small κ. When called for some $\phi = (s_0, ..., s)$ and $F'(\phi)$, the DPFP function first derives $F(\phi)$ from $F'(\phi)$ using constraint (3.11). It then iterates over all sets of functions $\{F(\phi, a) : a \in A(s)\}$

Algorithm 1 DPFP$(\phi = (s_0, ..., s), F'(\phi))$

1: $F(\phi)(t) \leftarrow \lfloor F'(\phi)(t)/\kappa \rfloor \cdot \kappa$
2: $u^* \leftarrow 0$
3: **for all** sets $\{F(\phi, a) : a \in A(s)\}$ s.t. $F(\phi)(t) = \sum_{a \in A(s)} F(\phi, a)(t)$ and $F(\phi, a)(t) = \kappa \cdot n; n \in N$ **do**
4: $\quad u \leftarrow 0$
5: $\quad$ **for all** $s' \in S$ **do**
6: $\quad\quad F' \leftarrow \sum_{a \in A(s)} P(s, a, s') \int_0^t F(\phi, a)(t') p_{s,a}(t - t') dt'$
7: $\quad\quad u \leftarrow u +$ **DPFP**$((s_0, ..., s, s'), F')$
8: $\quad$ **if** $u > u^*$ **then**
9: $\quad\quad$ BestSplitting $\leftarrow \{F(\phi, a) : a \in A(s)\}$
10: $\quad\quad u^* \leftarrow u$
11: **for all** $F(\phi, a) \in$ BestSplitting **and all** $t \in [0, \Delta]$ **do**
12: $\quad$ **if** $\lim_{\epsilon \to 0} F(\phi, a)(t + \epsilon) - F(\phi, a)(t) > 0$ **then**
13: $\quad\quad \widehat{\pi}^*(s)(t) \leftarrow a$
14: **return** $u^* + F(\phi)(\Delta) \cdot R(s)$

in order to find the best splitting of $F(\phi)$ *(lines 3—10)*. For a particular splitting, the DPFP function first makes sure that this splitting satisfies constraints (3.8) and (3.12) *(line 3)* upon which it calculates the total expected utility u of this splitting *(lines 4—7)*. To this end, for all paths $(s_0, ..., s, s')$, it uses constraint (3.10) to create functions $F' = F'((s_0, ..., s, s'))$ *(line 6)* and then, calls itself recursively for each pair $((s_0, ..., s, s'), F')$ *(line 10)*. Finally, if u is greater than the total expected utility u^* of the best splitting analyzed so far, DPFP updates the BestSplitting *(lines 8—10)*.

Upon finding the BestSplitting, the DPFP function uses Equation (3.6) to extract the best *deterministic* policy $\widehat{\pi}^*$ from it *(lines 11—13)* and terminates returning u^* plus the expected reward for entering s before time Δ (computed in line 14 by multiplying the immediate reward $R(s)$ by the probability $F(s_0, ..., s)(\Delta)$ of entering s before time Δ). As soon as DPFP$((s_0), 1)$ terminates, the algorithm extrapolates its already known point-based policies onto time interval $[0, \Delta]$ using the following method: If $\widehat{\pi}^*(s)(t_1) = a_1, \widehat{\pi}^*(s)(t_2) = a_2$, and $\widehat{\pi}^*(s)(t)$ is not defined for $t \in (t_1, t_2)$,

the algorithm puts $\widehat{\pi}^*(s)(t) = a_1$ for all $t \in (t_1, t_2)$. For example, if splitting in Figure 3.4 is optimal, $\widehat{\pi}^*(s_1)(t) = a_1$ for $t \in [0, t_1) \cup [t_1, t_2) \cup [t_2, t_3) \cup [t_4, \Delta)$ and $\widehat{\pi}^*(s_1)(t) = a_2$ for $t \in [t_3, t_4)$.

3.2.4 Taming the Algorithm Complexity

As stated, the DPFP algorithm can appear to be inefficient since it operates on large number of paths (exponential in the length of the longest path) and large number of splittings per path (exponential in $\lfloor 1/\kappa \rfloor$). However, this exponential complexity is alleviated thanks to the following features of DPFP:

- Varying policy expressivity for different states: The smaller the probability of traversing a path $\phi = (s_0, ..., s)$ before the deadline, the less expressive the policy for state s has to be (fewer ways in which $F(\phi)$ can be split into $\{F(\phi, a) : a \in A(s)\}$). For example, state s_2 in Figure 3.4 is less likely to be visited than state s_1 and therefore, DPFP allows for higher policy expressivity for state s_1 (2^4 policies) than for state s_2 (2^2 policies). Sparing the policy generation effort in less likely to be visited states enables faster policy generation.

- Varying policy expressivity for different time intervals: The smaller the probability of traversing to a state inside a time interval, the less expressive the policy for this state and interval has to be. In Figure 3.4 it is more likely to transition to state s_1 at time $t \in [t_1, t_3]$ (with probability 2κ) than at time $t \in [t_3, t_4]$ (with probability 1κ) and thus, DPFP considers 2^2 policies for time interval $[t_1, t_3]$ and only 2^1 policies for time interval $[t_3, t_4]$.

- Path independence: When function $F(\phi)$ for a sequence $\phi = (s_0, ...s)$ is split into functions $\{F(\phi, a) : a \in A(s)\}$, functions $\{F(\phi, a) : a \in A(s); P(s, a, s') = 0\}$ have no impact on

$F((s_0, ..., s, s'))$. Thus, fewer splittings of $F(\phi)$ have to be considered to determine all possible functions $F((s_0, ..., s, s'))$. For example, in Figure 3.4, $F((s_0, s_1, s_2))$ is only affected by $F((s_0, s_1), a_1)$. Consequently, as long as $F((s_0, s_1), a_1)$ remains unaltered when iterating over different splittings of $F((s_0, s_1))$, the best splittings of $F((s_0, s_1, s_2))$ does not have to be recomputed.

- Path equivalence. For *different* paths $\phi = (s_0, ..., s)$ and $\phi' = (s_0, ..., s)$ that coalesce in state s, the best splitting of $F(\phi)$ can be reused to split $F(\phi')$ provided that $\max_{t\in[0,\Delta]} |F'(\phi)(t) - F'(\phi')(t)| \leq \kappa$.

3.2.5 Error Control

Recall that $\widehat{F}^*$ is the optimal solution to the *restricted* dual problem returned by DPFP. It will now be proven that the reward error ϵ of a policy identified by $\widehat{F}^*$ can be expressed in terms of κ. To this end, it will first be proven that the maximum loss of probability mass for one sequence ϕ is bounded. Then, the error bound of the DPFP algorithm will be expressed as the sum of rewards collected on all possible execution sequences multiplied by the maximum loss of probability mass for one sequence.

Hence, it is first proven that for all paths $\phi \in \Phi(s_0)$:

$$\max_{t\in[0,\Delta]} |F^*(\phi)(t) - \widehat{F}^*(\phi)(t)| \leq \kappa|\phi| \tag{3.13}$$

Statement (3.13) is proven by induction on the length of ϕ.

- **Induction base:** For any sequence $\phi \in \Phi$ such that $|\phi| = 1$ it holds that $\max_{t\in[0,\Delta]} |F^*((s_0))(t) - \widehat{F}^*((s_0))(t)| = \max_{t\in[0,\Delta]} |1 - 1| = 0 < \kappa$.

- **Induction step**: Assume now that statement (3.13) holds for a sequence $\phi = (s_0, ..., s_{n-1})$ of length n. Statement (3.13) then also holds for all sequences $\phi' = (s_0, ..., s_{n-1}, s_n)$ of length $n + 1$ because

$$|F^*(\phi')(t) - \widehat{F}^*(\phi')(t)| \leq |F^*(\phi')(t) - \widehat{F'}^*(\phi')(t)| + \kappa$$

Where $\widehat{F}^*(\phi')$ is derived from $\widehat{F'}^*(\phi')$ using constraint (3.11)

$$\begin{aligned} &= \sum_{a \in A(s)} P(s, a, s') \Big| \int_0^t F^*(\phi, a)(t') \cdot p_{s,a}(t - t')dt' \\ &\qquad - \int_0^t \widehat{F}^*(\phi, a)(t') \cdot p_{s,a}(t - t')dt' \Big| + \kappa \\ &\leq \max_{a \in A(s)} \int_0^t |F^*(\phi, a)(t') - \widehat{F}^*(\phi, a)(t')| \cdot p_{s,a}(t - t')dt' + \kappa \\ &\leq \max_{a \in A(s)} \int_0^t |F^*(\phi)(t') - \widehat{F}^*(\phi)(t')| \cdot p_{s,a}(t - t')dt' + \kappa \end{aligned}$$

And from the induction assumption

$$\leq \int_0^t \kappa n \cdot p_{s,a}(t - t')dt' + \kappa \leq \kappa n + \kappa \leq \kappa |\phi'|$$

holds for $t \in [0, \Delta]$.

Consequently, statement (3.13) holds for any sequence $\phi \in \Phi$. The error ϵ of the DPFP algorithm can then be expressed by multiplying the maximum loss of probability mass for one sequence by the sum of rewards collected on all possible execution sequences. Precisely, error ϵ can be expressed in terms of κ because:

$$\begin{aligned} \epsilon &= R_{max} \sum_{\phi \in \Phi(s_0)} \max_{t \in [0,\Delta]} |F^*(\phi)(t) - \widehat{F}^*(\phi)(t)| \\ &\leq \kappa R_{max} \sum_{\phi \in \Phi(s_0)} |\phi| \leq \kappa R_{max} H |A|^H \end{aligned}$$

Where $R_{max} = \max_{s \in S} R(s)$ and H is the action horizon (if the minimal action duration δ is known than $H \leq \lfloor \Delta / \delta \rfloor$). Hence, by decreasing κ, DPFP can trade off speed for optimality.

Chapter 4: Experiments with Single Agent Algorithms

This chapter reports on the experimental evaluation of the algorithms for solving continuous resource MDPs. Section 4.1 demonstrates a feasibility study of the CPH algorithm introduced in Section 3.1. CPH is compared with the Lazy Approximation algorithm [Li and Littman, 2005], currently the fastest algorithm for solving continuous resource MDPs, on various configurations of the domain from Figure 2.1. Then, Section 4.2 reports on a comparison of CPH and Lazy Approximation efficiency on a family of computationally intensive planetary exploration domains — these two algorithms are joined by the DPFP algorithm in this comparison. Next, Section 4.3 reports on the empirical evaluation of the DPFP-CPH hybrid algorithm that combines the strengths of CPH and DPFP. Finally, Section 4.4 reports on the successful integration of CPH with RIAACT [Schurr et al., 2008], the adjustable autonomy system for coordinating a human incident commander and an agent team in an event of a simulated large scale disaster.

4.1 CPH Feasibility Experiments

This section reports on the small scale experiments involving CPH and Lazy Approximation [Li and Littman, 2005], currently the leading algorithm for solving continuous resource MDPs. Lazy Approximation approximates the probability distributions and value functions with piecewise

constant functions. The Bellman updates then transform the piecewise constant value functions into piecewise linear value functions, that then need to get approximated again with piecewise constant value functions. These repeated approximations result in large runtimes and approximation errors that are demonstrated below. Furthermore, the number of intervals needed to approximate the value functions with piecewise constant functions is large, which results in even large runtimes.

CPH, on the other hand, approximates the probability distributions with phase-type distributions, resulting in exponential probability distributions that it then uniformizes. One of its advantages is that the value functions then remain piecewise gamma functions and it thus does not need to approximate the value functions at all, with only one small exception, which involves finding the intersections of value functions, for which it uses a numerical method. Another one of its advantages is that the number of intervals of the piecewise gamma value functions tends to be smaller than the number of intervals of the piecewise linear value functions. Both of these advantages result in significant computational savings as can be seen below.

The first set of experiments compares the efficiency of CPH and Lazy Approximation (referred to as LA) for the planetary exploration domain introduced in Section 2.1.1. The planetary exploration domain might appear small with only 5 discrete states. However, if one uses standard MDP framework where time-to-deadline is discretized into 1/100 time units (which for a 4 hour period of rover operation corresponds to a decision point every 36 seconds), then there are already $5 * 400 = 2000$ distinct MDP states. For the three experiments (Figures 4.1, 4.2 and **??**) the error $\max_{0 \leq t \leq \Delta} |V^*(start)(t) - V(start)(t)|$ of the calculated value function $V(start)$ is always plot on the x-axis and the corresponding runtime (measured in milliseconds) in logarithmic scale on the y-axis.

Experiment 1 determines how CPH and Lazy Approximation trade off between runtime and error for the planetary exploration domain from Section 2.1.1. Since the action durations in the Mars rover domain are already distributed exponentially and thus phase-type with one phase, there are no errors introduced by approximating the probability distributions over the action durations with phase-type distributions. Also, since these distributions are equal, uniformization will not introduce self-transitions and value iteration can run for a finite number of iterations. Thus, for CPH, the accuracy of the bisection method (that determines the dominance breakpoints) was varied whereas for Lazy Approximation, the accuracy of the piecewise constant approximations of the probability distributions over the action durations and value functions was varied. The results show that CPH is faster than Lazy Approximation with the same error, by three orders of magnitude for small errors. For example, CPH needed 2ms and Lazy Approximation needed 1000ms to compute a policy that is less than 1% off optimal, which corresponds to an error of 0.13 in the Mars rover domain.

Experiments 2 and 3: determine how CPH and Lazy Approximation trade off between runtime and error when all action durations in the planetary exploration domain from Section 2.1.1 are characterized by either Weibull distributions $Weibull(\alpha = 1, \beta = 2)$ (Experiment 2) or Normal distributions $N(\mu = 2, \sigma = 1)$ (Experiment 3). Since the action durations are no longer exponential, phase-type approximation was used for CPH. Here, the accuracy of the bisection method used by CPH has been fixed, but the number of phases used by the Coxian distribution (see Section B) approximating the initial duration distribution was varied. As can be seen, CPH is still faster than Lazy Approximation with the same error, by more than one order of magnitude for small errors. For example, CPH needed 149ms (with five phases) and Lazy Approximation needed 4471ms to compute a policy that is less than 1% off optimal for the Weibull distributions.

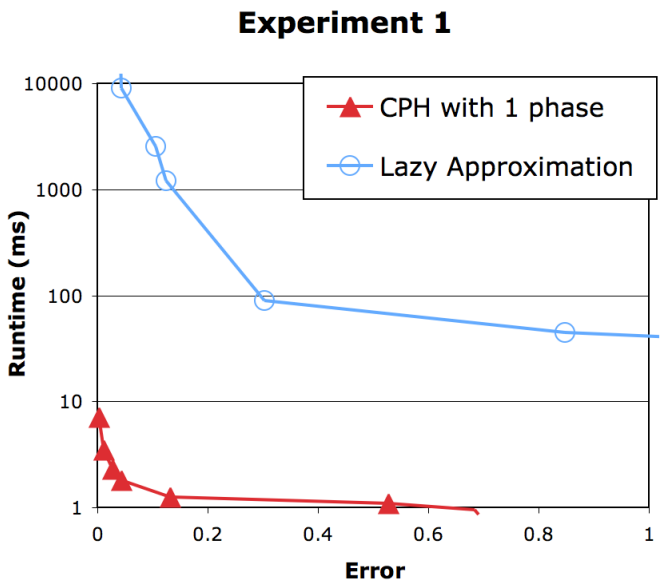

Figure 4.1: Empirical evaluation of CPH for exponential distributions.

In addition, the study the tightness of the error bound for CPH, calculated in Section 3.1.4 has been conducted. To this end, CPH was run with varying number of phases (2,3 and 5) used to approximate the probability distributions which in turn affected both the error ϵ and the unified exit rate parameter λ for the exponential distributions. Those numbers were then plugged into the formula from Theorem 1 in order to calculate the theoretical planning horizon for CPH. As can be seen (Figure 4.3) CPH converged much faster than the theoretical planning horizon calculated from Theorem 1. That encouraging results suggests that tightening the error bound for CPH can be a worthy topic of future investigation.

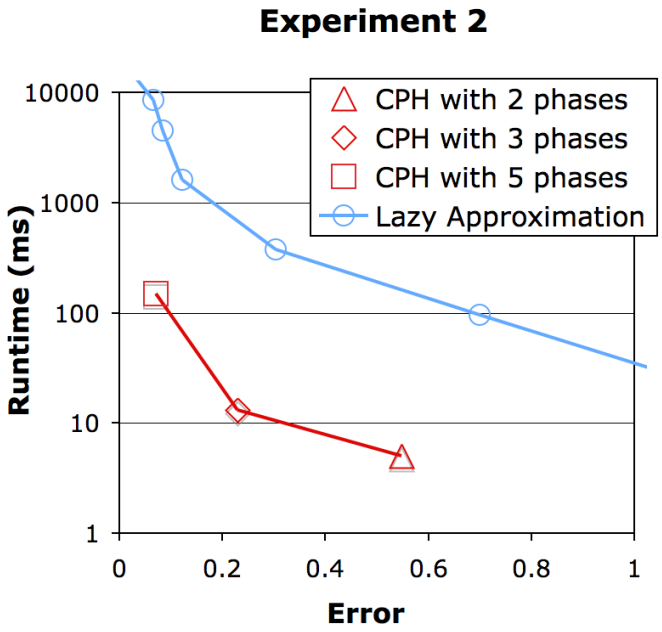

Figure 4.2: Empirical evaluation of CPH for Weibull distributions.

Number of phases	Error	Planning horizon	Theoretical planning horizon
5	0.11	35	106422
3	0.33	21	1823
2	0.87	14	264

Figure 4.3: Planning horizon of CPH.

4.2 CPH and DPFP Scalability Experiments

This section reports on the scalability experiments of CPH, DPFP and the Lazy Approximation algorithms. The domains on which all three algorithm were run, the fully ordered domain, unordered domain and partially ordered domain, are presented in Figures 4.4a, 4.4b, 4.4c. In the fully ordered domain the agent executes an action $a' \in A(s_0) = \{a_1, a_2, a_3\}$, transitions to state $s_{a'}$, executes $a'' \in A(s_{a'}) = \{a_1, a_2, a_3\}$, transitions to state $s_{a',a''}$ — it repeats this scheme up to

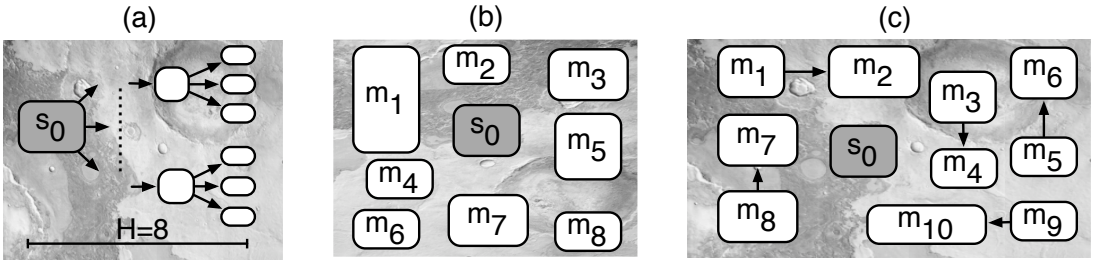

Figure 4.4: Domains for CPH and DPFP scalability experiments: m_i denote sites of interest

$H = 8$ times for a total number of $3^8 = 6561$ states. In the unordered domain (which resembles the classical Traveling Salesman Problem) the agent visits up to 8 sites in an arbitrary order and hence, the number of states is $2^8 = 256$. Finally, the partially ordered domain in a combination of the fully and unordered domains; here, the agent can visit up to 10 sites in a partial order, that is, site $m + 1$ can be visited only after site m has already been visited for $m = 1, 3, 5, 7, 9$. For all the domains, the mission deadline $\Delta = 10$. Also, action rewards are drawn uniformly from set $\{1, 2, ..., 10\}$ and action durations are sampled from one of the following probability distribution functions (chosen at random): *Normal*($\mu = 2$,$\sigma = 1$), *Weibull*($\alpha = 2$,$\beta = 1$), *Exponential*($\lambda = 2$) and *Uniform* ($a = 0$,$b = 4$).

The scalability experiments that have been conducted determine how the three algorithms trade off between runtime and error. The following parameters were varied:

- For DPFP, the height κ of the step function that approximates the probability functions;
- For CPH, the number of phases of the Coxian distribution used to approximate the initial action duration distributions.
- For Lazy Approximation, the tolerance threshold used when approximating piecewise linear functions with piecewise constant functions in Bellman updates.

The results of the scalability experiments are shown in Figures 4.5a, 4.5b, 4.5c where runtime (in seconds) is on the x-axis (notice the logarithmic scale) and the solution quality (% of the optimal solution) is on the y-axis. In particular, to obtain the benchmark optimal solution quality, Lazy Approximation was run sufficiently long so that the error margin of its solution was below 10% of the optimal solution (to see the detailed control of this error, refer to [Li and Littman, 2005]). The results across all the domains show CPH or DPFP outperform Lazy Approximation in terms of runtimes necessary to find high quality solutions by up to three orders of magnitude. For example, to find a solution that is less than 10% off the optimal policy for the unordered domain, CPH needs 14.5s whereas Lazy Approximation requires 2026.1s for the same task.

Furthermore, as can be seen, DPFP opens up an entirely new area of the solution-quality vs time tradeoff space that was inaccessible to previous algorithms. In particular DPFP dominates Lazy approximation in this tradeoff, providing higher quality in lower time. DPFP also provides very high quality an order of magnitude faster than CPH, e.g. in Figure 4.5a for solutions with quality higher than 70%, DPFP will provide an answer in 0.46 seconds, while CPH will take 28.1s for the same task. Finally, DPFP exhibits superior anytime performance, e.g. in Figure 4.5c, run with $\kappa = 0.3, 0, 25, 0.2$ it attains solution qualities $42\%, 61\%, 72\%$ in just $0.5s, 1.1s, 3.9s$.

4.3 DPFP-CPH Hybrid Experiments

Encouraged by DPFP's any-time performance and CPH's superior quality results, a DPFP-CPH hybrid algorithm was developed and its efficiency was contrasted with the efficiency of a stand-alone CPH. The idea of the DPFP-CPH hybrid is to first use DPFP to quickly find an action to be executed from the starting state s_0 and then, use this action to narrow down CPH's search for

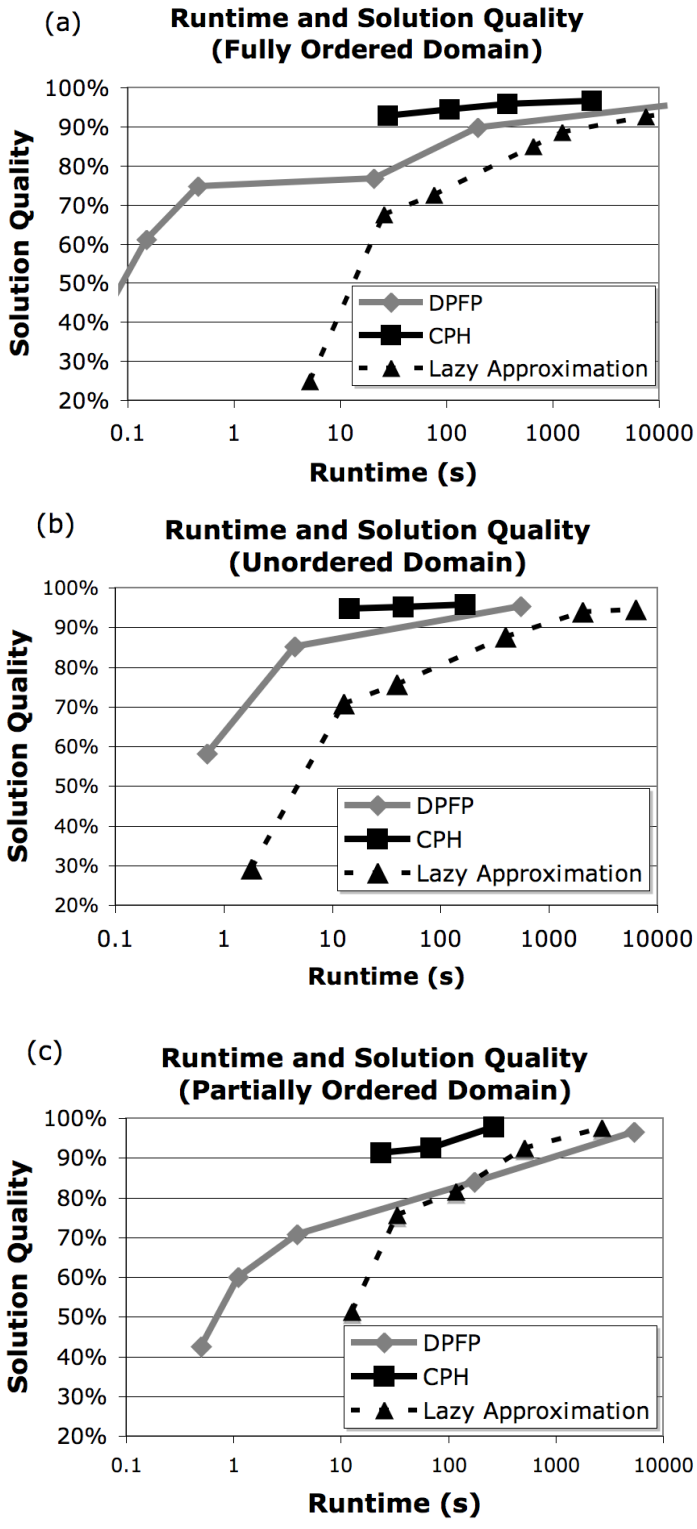

Figure 4.5: Scalability experiments for DPFP, CPH and Lazy Approximation

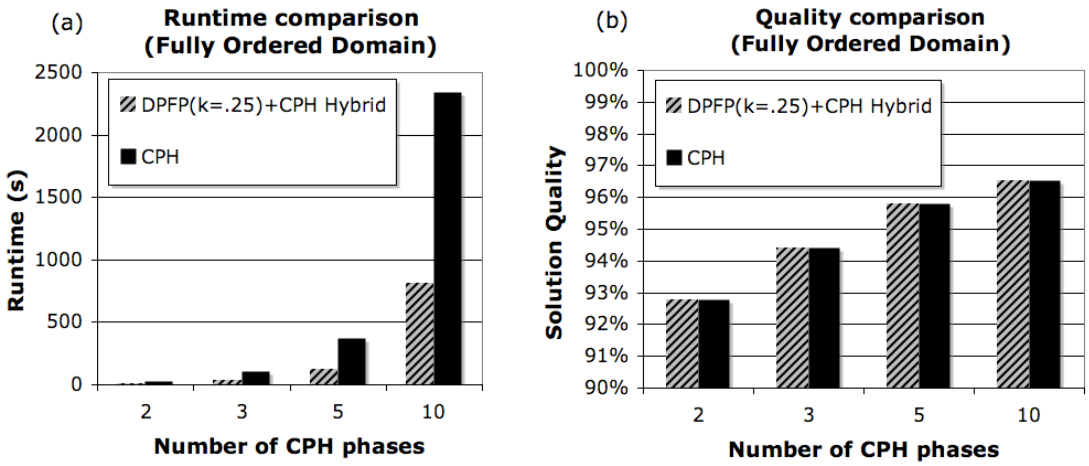

Figure 4.6: DPFP+CPH hybrid: Fully ordered domain

a high quality policy. For example, the hybrid that was implemented uses DPFP (with $\kappa = .2$) to suggest to CPH which action should be executed in s_0 at time 0, and then runs CPH in the narrowed state-space.

The empirical evaluation of the DPFP-CPH hybrid is shown in Figures 4.6, 4.7 and 4.8. Across all the Figures, the accuracy of CPH is varied on the x-axis (using more phases translates into higher accuracy of CPH) whereas the y-axis is used to plot the algorithms runtime (in Figures 4.6a, 4.7a, 4.8a) and the corresponding solution quality (in Figures 4.6b, 4.7b, 4.8b). The results across all the domains show that the DPFP-CPH hybrid attains the same quality as stand-alone CPH, yet requires significantly less runtime (over 3 times). For example, when CPH accuracy is fixed at 5 phases, DPFP-CPH hybrid needs only 51s to find a solution whereas stand-alone CPH needs 169.5s for the same task (Figure 4.7a).

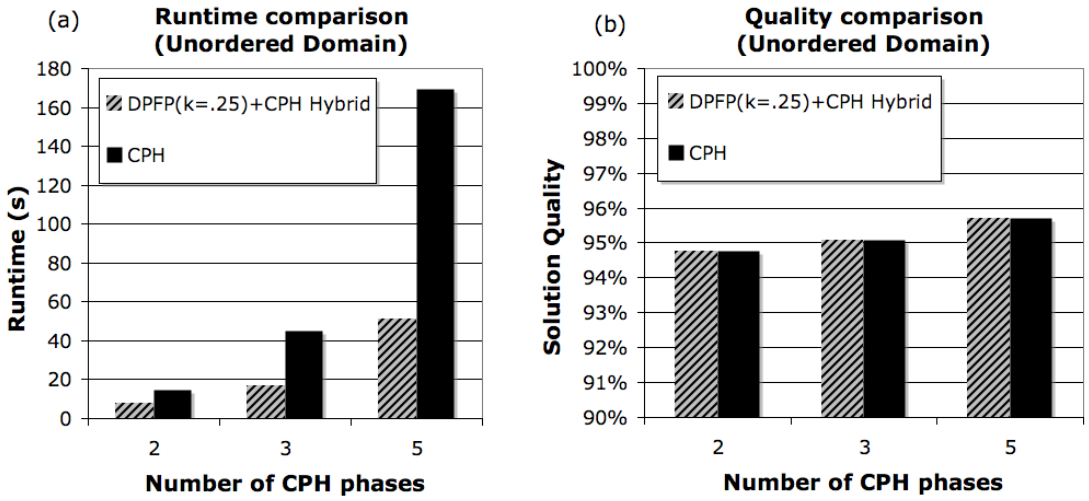

Figure 4.7: DPFP+CPH hybrid: Unordered domain

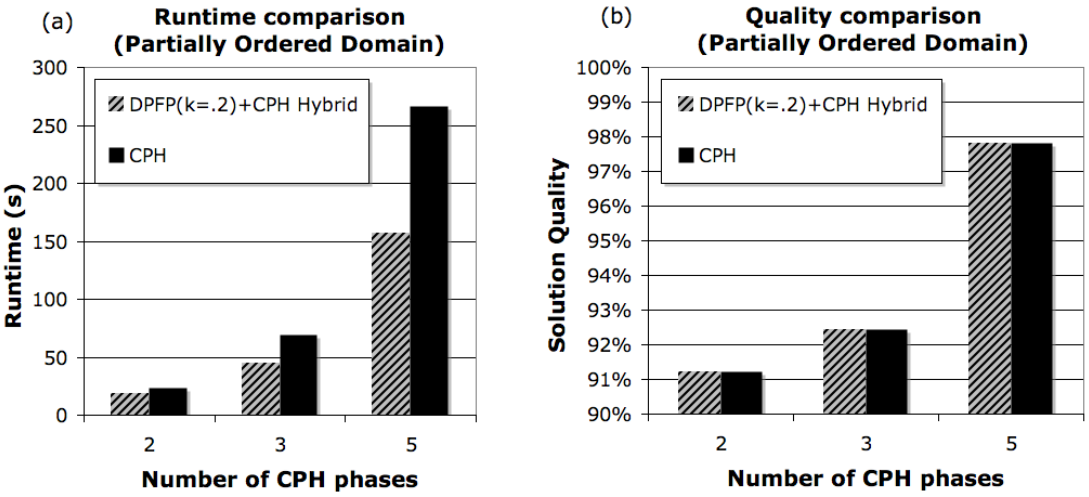

Figure 4.8: DPFP+CPH hybrid: Partially ordered domain

4.4 DEFACTO Experiments

In the final set of experiments with single agent algorithms, CPH was integrated with RIAACT [Schurr et al., 2008] - the adjustable autonomy module of the DEFACTO system [Schurr et al., 2005] for training the incident commanders of the Los Angeles Fire Department. The main

purpose of RIAACT is to decide who should perform the role allocation in an even of a simulated city wide disaster: Either (i) the human incident commander or (ii) the agent team. In essence, whenever a new role appears, e.g. whenever a new building goes on fire, either the human incident commander or the agent team must decide which fire engine should fight that particular fire. In this context, it has been demonstrated [Marecki et al., 2005] that, preventing decision conflicts between the human incident commander and the agent team is a fundamental problem, as it has direct implications on the outcome of the disaster rescue operation.

At a basic level, RIAACT is an instantiation of the continuous resource MDP model. RIAACT makes the following assumptions: (i) Time is a continuous resource; (ii) time at which a building is completely burnt is the resource limit and (iii) the action durations are uncertain. Precisely, RIAACT's states are $S = \{Aa, Ha, Adi, Adc, Hdi, Hdc, Finish\}$ with the following meaning (see Figure 4.9):

- *Aa*: The agent team is responsible for deciding who should execute the current role;
- *Ha*: The human incident commander is responsible for deciding who should execute the current role;
- *Adi*: Agent team has decided who should execute the current role, but this decision is inconsistent with the human incident commander's preferences;
- *Adc*: Agent team has decided who should execute the current role and this decision is consistent with the human incident commander's preferences;
- *Hdi*: The human incident commander has decided who should execute the current role, but this decision is inconsistent with the preferences of the agent team.

- Hdc: The human incident commander has decided who should execute the current role and this decision is consistent with the preferences of the agent team.

- $Finish$: The role allocation has been performed

RIAACT's actions $A = \{TransferAutonomy, Decide, Resolve, Execute\}$ are as follows:

- $TransferAutonomy$: At any point in time, the agent team can transfer the autonomy (role allocation request) to the human incident commander (similarly, the human incident commander can transfer the autonomy to the agent team). Although the $TransferAutonomy$ action does not yield any reward, it consumes a certain amount of time which is sampled from a given probability distribution.

- *Decide*: At any point in time, when the role allocation request is *at* the human incident commander (in state *Aa*), the human incident commander can make a role allocation decision. Depending on whether this decision is consistent or not with the agent team, the MDP transitions to state *Hdi* or state *Hdc* (similarly, when the agent team makes a decision, the MDP transitions from state *Aa* to either state *Adi* or state *Adc*). The duration of the *Decide* action is uncertain and follows a given probability distribution. RIAACT assumes that the human incident commander takes longer than the agent team to make a decision. The *Decide* action itself yields no reward.

- *Resolve*: When the role allocation decision is inconsistent (the process transitioned to an inconsistent state under the *Decide* action), a *Resolve* action can be performed, to improve the decision so that it becomes consistent. The *Resolve* action does not come for free — it consumes a certain amount of time which is sampled from a given probability distribution.

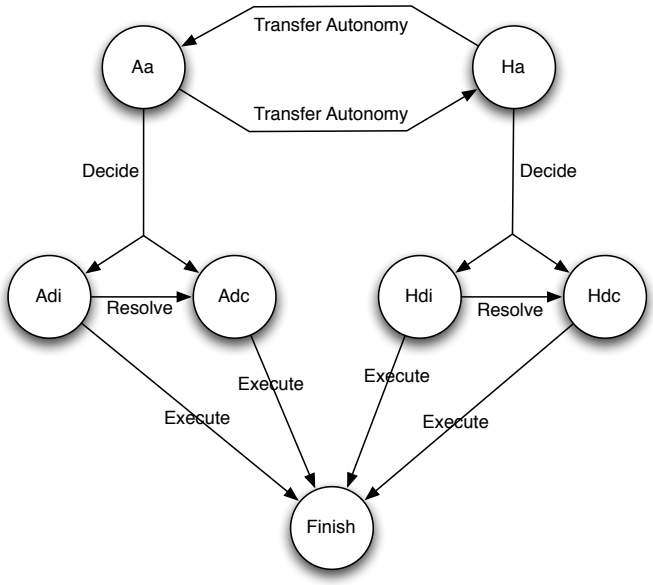

Figure 4.9: RIAACT: Adjustable autonomy component of the DEFACTO system, [Schurr et al., 2008]

Although there is no reward for the *Resolve* action, consistent role allocations yield more reward when the *Execute* action is performed (see below).

- *Execute*: When the *Execute* action is performed, the current role is executed by some entity. Depending on whether the process occupies states Adi, Adc, Hdi or Hdc when the *Execute* action is performed, different rewards can be earned. Typically, the highest reward is earned if the *Execute* action is started from state *Hdc* (the human incident commander decides who should execute a role, and this decision is consistent with the agent team) and the lowest reward is earned if the *Execute* action is started from state *Adi* (the agent team decides who should execute a role, but this decision is inconsistent with the intention of the human incident commander). The duration of the *Execute* action is uncertain; it is affected by the intensity of the fire and the proximity of the fire engines.

To solve the RIAACT model shown in Figure 4.9 the CPH algorithm was used. CPH employed 3-phase Coxian distributions for the purpose of the underlying phase-type approximation. The CPH solver was run for $n^* = 80$ Bellman update iterations, to ensure that the increase if value functions in further iterations was marginal. After 27.2 seconds, CPH returned a time dependent policy, both for the human incident commander as well as the agent team, for any state present in the RIAACT model. As a result, depending on the time remaining before a building is completely burnt, the human incident commander and the agent knew exactly whether it is more profitable to: (i) Make a role allocation decision and then try to resolve a potential conflict or (ii) Choose to transfer the autonomy. The RIAACT approach was compared with the two competing approaches: (i) The "Always Reject" approach in which the agent team would always perform a no-return transfer of autonomy to the human incident commander and the (ii) "Always Accept" approach in which the agent team would always accept a role allocation request and never bother the human incident commander.

The experimental results of DEFACTO equipped with RIAACT are demonstrated in Figure 4.10. Here, the number of buildings saved in plotted on the y-axis (averaged over 50 runs) whereas the duration of the *Resolve* action (which is sampled from a Normal distribution with a different mean and variance values) is varied on the x-axis. As can be seen, RIAACT approach outperforms the "Always Accept" and "Always Reject" approaches for any tested distribution of the *Resolve* action duration. In particular, when the *Resolve* action duration follows a normal distribution with a mean of 3s and the variance of $1s^2$, DEFACTO with RIAACT allows to save up to 25% more building in an event of a simulated disaster rescue operation. Furthermore, the duration of the *Resolve* action does not have an impact on the number of building saved for the "Always

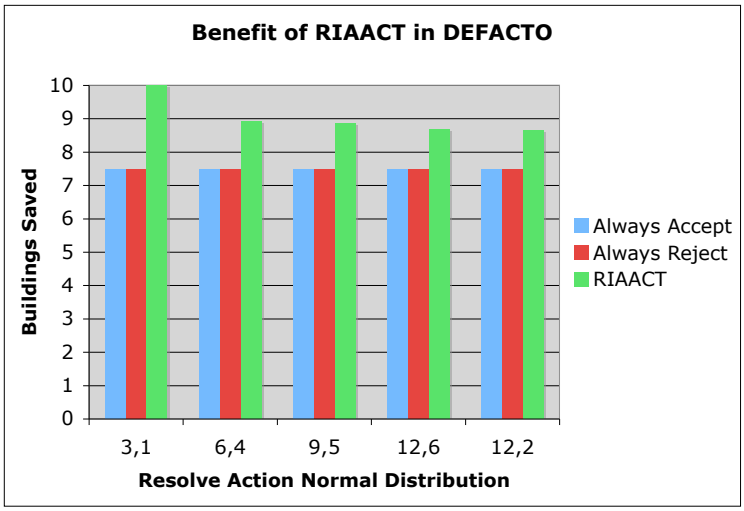

Figure 4.10: RIAACT results, [Schurr et al., 2008]

Accept" and "Always Reject" approaches — this is not surprising because these strategies do not use RIAACT's inconsistency resolution mechanism.

Chapter 5: Multiagent Solutions

This chapter focuses on techniques for solving planning problems modeled as Continuous Resource, Decentralized MDPs (CR-DEC-MDPs). Two different algorithms for solving CR-DEC-MDPs are proposed. The first algorithm (VFP) considers a special case when CR-DEC-MDPs are fully ordered (explained later). For such CR-DEC-MDPs, VFP performs a series of policy iterations to quickly find locally optimal joint policies. In addition, VFP implements a set of heuristics aimed at improving the quality of these locally optimal joint policies. The second proposed algorithm (M-DPFP) for solving CR-DEC-MDPs operates on arbitrary CR-DEC-MDPs. It finds joint policies that are guaranteed to be within an arbitrary small ϵ from the optimal joint policies by leveraging the concept of probability function propagation to a multi-agent setting. Both algorithms consider $AbsoluteTime$ as a continuous resource whose values monotonically *increase* from 0 to some mission deadline Δ[1].

5.1 Locally Optimal Solution: The VFP Algorithm

Because CR-DEC-MDPs are hard to solve optimally, this section introduces a locally optimal algorithm (VFP) that operates on CR-DEC-MDPs that are fully-ordered. The approach to simplify

[1]Such formulation of a continuous resource does not affect the planning problems because $\Delta - AbsoluteTime$ is simply a continuous resource whose values monotonically decrease

the problem by restricting the search for policies to fully ordered Decentralized MDP has been first proposed in [Beynier and Mouaddib, 2005] and then elaborated on in [Beynier and Mouaddib, 2006]. In these works, the authors developed a locally optimal algorithm that has been shown to scale up to domains with double digit action horizons. The VFP algorithm proposed in this section builds on on the idea to perform a search for locally optimal policies to fully-ordered CR-DEC-MDPs.

Precisely, a fully-ordered CR-DEC-MDP assumes that methods are arranged in chains, i.e., for any agent n and its set of methods $M_n = \{m_1, ..., m_k\}$ there exist resource precedence constraints $\langle i, i+1 \rangle \in C_{\prec}$ for all $i = 1, ..., k-1$ which impose a chain ordering of methods from M_n. Two immediate facts result from imposing such restrictions on CR-DEC-MDPs:

- When agent n executes method m_i successfully, it can either start executing method m_{i+1} (this action is denoted as E) or choose to remain idle for a certain amount of time and then make the decision (the action is denoted as W);

- When agent n executes method m_i unsuccessfully, it can no longer execute any other method because none of the methods $m_{i+1}, m_{i+2}, ..., m_k$ that directly or indirectly precede method m_i will ever be enabled. Hence, the execution of agent n stops. The execution of other agents can continue, provided that that all their methods have been executed successfully thus far.

When operating on fully-ordered CR-DEC-MDPs, agent policy for a method m_{i+1} is directly related to the expected utility of starting the execution of m_{i+1} as shown in Figure 5.1. Here, the agent is about to start executing a method m_i which can be executed in one of the two time windows (resource ranges): $\langle i, l_1, l_2 \rangle \in C_{[]}$ or $\langle i, l_3, l_4 \rangle \in C_{[]}$. The shaded area represents the

total expected utility for starting the execution of method m_i over time (referred to as the *value function*). As can be seen in Figure 5.1, at any time $t \in [0, \Delta]$, the **E**xecute action is better than the **W**ait action if it is less profitable to perform the **E**xecute action in the future (after time t).

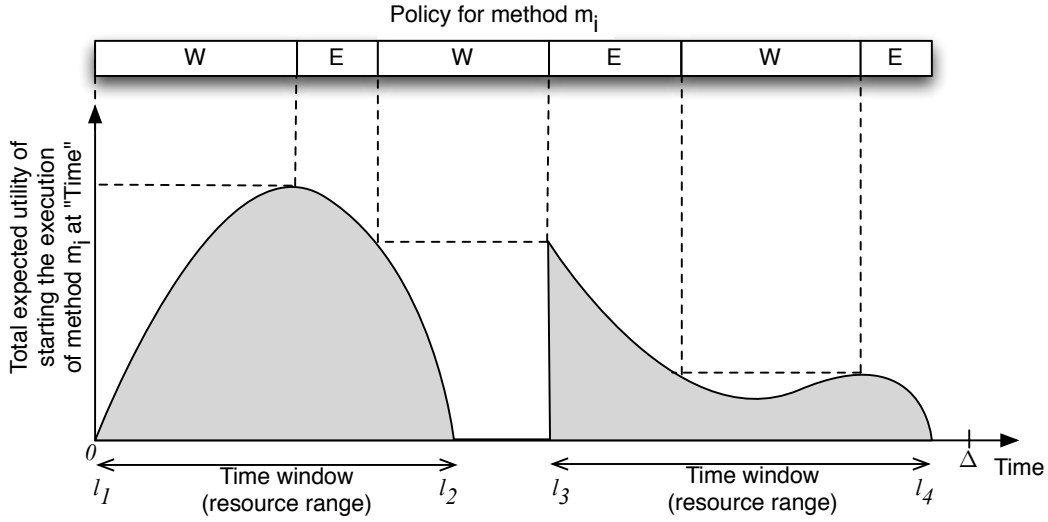

Figure 5.1: Agent policy in a fully-ordered CR-DEC-MDPs

Since in a fully-ordered CR-DEC-MDP each agent n always knows which method should be executed next, agent decision making is reduced to choosing the correct method execution starting time. However, since the model allows for many resource limit constraints for a method m_i (multiple execution time windows in Figure 5.1), the policy π_n of agent n cannot be stored as a set of pairs (m_k, t_k) where t_k is a point in time when the agent switches from action W to action E. Instead, the policy π_n of agent n must be a function $\pi_n : M_n \times [0, \Delta] \mapsto \{W, E\}$ where "W" represents the Wait action and "E" represents the Execute-next-method action. For example, $\pi_n(\langle i, t \rangle) = E$ means that, at time t, when agent's next method to be executed is m_i, it will start executing it. Else, if $\pi_n(\langle i, t \rangle) = W$ the agent will wait for an infinitesimal amount of time ϵ and

then make another decision at time $t + \epsilon$. In the following, the expression $\pi_n(\langle i, t\rangle)$ is referred to as a policy of agent n for method m_i at time t.

5.1.1 Policy Iteration Approach

A popular approach to find the optimal joint policy π^* is to use the value iteration principle which in the context of CR-DEC-MDP could work as follows: In order to determine the optimal policy for method m_i one can propagate backwards (in the inverse direction to the resource precedence constraints relation) the expected utilities of executing methods m_j where $\langle i, j\rangle \in C_{\prec}$. Unfortunately, for the CR-DEC-MDP model, the optimal policy for method m_i also depends on policies for methods m_j where $\langle j, i\rangle \in C_{\prec}$. This bi-directional dependency results from the fact that the expected reward for starting the execution of method m_i at time t also depends on the probability that method m_i will be enabled before time t. Consequently, as shown in [Bernstein et al., 2000], the complexity of the optimal algorithms for a CR-DEC-MDP model with discrete resource levels is NEXP-complete.

Following the limited applicability of globally optimal algorithms, locally optimal algorithms have recently gained a lot of attention. Of particular importance is the Opportunity Cost, Decentralized MDP (OC-DEC-MDP) algorithm [Beynier and Mouaddib, 2005], [Beynier and Mouaddib, 2006] that has been shown to scale up to problems involving double digit action horizons. The idea of the OC-DEC-MDP algorithm, which also operates on fully ordered MDPs, is to perform a series of policy iterations to converge on a locally optimal solution. The algorithm starts with the *earliest starting time* policy according to which an agent starts executing a method m_i as soon as it possible. It then improves this policy by performing a *opportunity cost propagation*

phase, evaluation the new policy and then, if the new policy is significantly better than the old policy, the *probability propagation phase* which prepares the algorithm for its next iteration.

The opportunity cost and probability propagation phases of the OC-DEC-MDP algorithm consider the *AbsoluteTime* as a resource, but a discretized one. The algorithm then operates on discrete time intervals: $F_i[t_1, t_2]$ is the probability that method m_i will be executed in time interval $[t_1, t_2]$, and $O_i[t_1, t_2]$ is the expected utility for executing method m_i in time interval $[t_1, t_2]$ assuming that method m_i is enabled — referred to as the *Opportunity Cost* of method m_j in time interval $[t_1, t_2]$. At each iteration the OC-DEC-MDP algorithm knows the old policy π and the probabilities $F_i[t_1, t_2]$ for $t_1, t_2 \in \mathbb{N} \cap [0, \Delta]$ uniquely identified by π. The algorithm then searches for a new policy π' that improves the old policy π in the following two phases:

- **Opportunity cost propagation**: It calculates opportunity costs $O_i[t_1, t_2]$ for $t_1, t_2 \in \mathbb{N} \cap [0, \Delta]$ and $m_i \in M$ by backward propagation starting from *sink methods* (methods that do not enable any other methods) and ending on *source methods* (methods not enabled by any other method). Opportunity cost propagation phase utilizes the probabilities $F_i[t_1, t_2]$ uniquely identified by the old policy π.

- **Probability propagation**: Using the opportunity costs $O_i[t_1, t_2]$ calculated in the opportunity cost propagation phase, the algorithm identifies the most profitable method execution intervals which are then used to determine the new policy π'. The algorithm then calculates the probabilities $F_i[t_1, t_2]$ for $t_1, t_2 \in \mathbb{N} \cap [0, \Delta]$ and $m_i \in M$ associated with the new policy π'. To this end, it propagates the probabilities $F_i[t_1, t_2]$ forward (from source methods to sink methods). If the new policy π' does not improve the old policy π by more than some

margin ϵ, the OC-DEC-MDP algorithm terminates. Otherwise, the algorithm repeats the two above-mentioned steps.

5.1.2 Functional Representation

Unfortunately, the OC-DEC-MDP algorithm has three major shortcomings: (i) It is only applicable to solving problems that assume discrete resource levels; (ii) it does not exploit the functional representation of the underlying opportunity cost functions and probability functions and can thus run slow and (iii) it fails to address a critical problem of double-counting when the opportunity costs $O_i[t_1, t_2]$ are being derived from opportunity costs $O_j[t_1, t_2]$ for $\langle i, j \rangle \in C_{\prec}$.

To remedy these shortcomings, the VFP algorithm for solving fully-ordered CR-DEC-MDPs is proposed. VFP borrows from OC-DEC-MDP the idea to perform a series of policy improvement phases to find a locally optimal solution. Yet, VFP addresses the above-mentioned shortcomings of the OC-DEC-MDP algorithm: First, VFP maintains and manipulates opportunity cost functions and probability functions over time for each method rather than discrete opportunity costs and probabilities for each pair of method and time interval. Such representation allows VFP to group the time points for which the opportunity cost function changes at the same rate which translates into significant speedups of policy improvement phases. Second, VFP's functional representation preserve the structure of the underlying opportunity cost functions which allows VFP to identify and correct the critical overestimations of the expected utilities of method execution.

The general scheme of the VFP algorithm is identical to the OC-DEC-MDP algorithm — both algorithms perform a series of policy improvement iterations. However, instead of propagating opportunity costs and probabilities in each iteration, VFP propagates opportunity cost

functions and probability functions. These two phases are therefore referred to as the *opportunity cost function propagation* and the *probability function propagation*. In order to implement these phases using functional representation, for each method $m_i \in M$, the VFP algorithm employs three different functions (related to each other in Equation 5.4):

- **Opportunity Cost Function** $O_i(t)$ maps time $t \in [0, \Delta]$ to the expected utility for starting the execution of method m_i at time t assuming that m_i is enabled (assuming that the execution of methods m_j such that $\langle i, j \rangle \in C_{\prec}$ has been finished successfully before time t).

- **Probability Function** $F_i(t)$ maps time $t \in [0, \Delta]$ to the probability that method m_i will be successfully executed *before* time t.

- **Value Function** $V_i(t)$ maps time $t \in [0, \Delta]$ to the expected utility for starting the execution of method m_i at time t.

The unique role of the value function $V_i(t)$ is to extract the current policy. Precisely, the policy $\pi_n(\langle i, t \rangle)$ of agent n for method m_i at time t is calculated as follows (see Figure 5.1):

$$\pi_n(\langle i, t \rangle) = \begin{cases} W & \text{if there exists } t' > t \text{ such that } V_i(t') > V_i(t) \\ E & \text{otherwise.} \end{cases}$$

Where **W** is the wait action and **E** is the execute method action (informally, the agent will **W**ait if it is more profitable to start the execution of method m_i later).

The next two sections demonstrate how VFP performs a single policy iteration. To this end, it is first shown how the analytical operations update the opportunity cost functions. It is then shown how the opportunity cost functions are used to determine the value functions and how to determine the policy using the value functions. Finally, it is shown how the analytical operations

update the probability functions — a step necessary to prepare the VFP algorithm for its next policy iteration.

5.1.3 Opportunity Cost Function Propagation Phase

In analogy to [Beynier and Mouaddib, 2005], [Beynier and Mouaddib, 2006], the opportunity function propagation phase consists in propagating the opportunity cost functions through the graph $G = (M, C_{\prec})$ of methods M linked by resource precedence constraints $C_{\prec}$, starting from the sink methods to the source methods. In particular, for each method m_{i_0} encountered during this phase (refer to Figure 5.3), the opportunity cost function O_{i_0} is calculated from the opportunity cost functions O_{j_n} of methods m_{j_n} that follow method m_{i_0} in graph G.

Let $n = 0, 1, ..., N$ as shown in Figure 5.3. The opportunity cost function $O_{i_0}(t)$ for time $t \in [0, \Delta]$ is then derived as follows (refer to Equation 5.1): If there is no time window $\langle i_0, t_1, t_2 \rangle \in C_{[]}$ for time t such that $t_1 \leq t \leq t_2$, the agent will not start the execution of method m_{i_0} because there is no chance that the execution of method m_{i_0} will be successful and hence, $O_{i_0}(t) = 0$. Otherwise, the execution of method m_{i_0} can be started, because there is a non-zero chance that this execution will be successful (completed before time t_2). In this case, with probability $p_{i_0}(t')$, the execution of method m_{i_0} will take time t' to complete and as long as $t + t' \leq t_2$ (see Figure 5.2), method m_{i_0} will be executed successfully (provided that method m_{i_0} was enabled at time t). Here, the agent will earn the immediate reward r_{i_0} plus $\sum_{n=0}^{n=N} O_{j_n,i_0}(t + t')$ where $O_{j_n,i_0}(t + t')$ is the opportunity cost that method m_{i_0} will receive for enabling the execution of method m_{j_n} at time $t+t'$. Functions $O_{j_0,i_k}(t + t')$ for $k = 0, 1, ..., K$ are called *splittings* of the opportunity cost function O_{j_0} and it is

temporally assumed here that $O_{j_n,i_0} := O_{j_n}$ — this assumption is waived in Section 5.1.5. The above discussion justifies why the opportunity cost $O_{i_0}(t)$ is calculated by:[2]

$$O_{i_0}(t) = \begin{cases} \int_0^{t_2-t} p_{i_0}(t')(r_{i_0} + \sum_{n=0}^{N} O_{j_n,i_0}(t+t'))dt' & \text{if } \exists \langle i_0, t_1, t_2 \rangle \in C_{[]} \text{ s.t. } t \in [t_1, t_2] \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

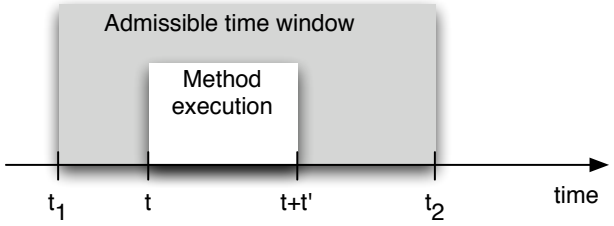

Figure 5.2: Method execution occurring within an admissible time window

It is now shown how to derive O_{j_0,i_0} (derivation of O_{j_n,i_0} for $n \neq 0$ follows the same scheme). Let $\overline{O}_{j_0,i_0}(t)$ be the opportunity cost of starting the execution of method m_{j_0} at time t assuming that method m_{i_0} has been completed. It can be derived by multiplying O_{j_0} by the probability functions for all methods other than m_{i_0} that enable m_{j_0}, that is:

$$\overline{O}_{j_0,i_0}(t) = O_{j_0}(t) \cdot \prod_{k=1}^{K} F_{i_k}(t). \tag{5.2}$$

Note, that this derivation is only approximate because in general the probability functions $\{F_{i_k}\}_{k=1}^{K}$ do not have to represent independent random variables (similar approximation is used in [Beynier and Mouaddib, 2005], [Beynier and Mouaddib, 2006]). Now, the opportunity cost

[2]Note, that the calculation of $O_{i_0}(t)$ is equivalent to the convolution operation, because for $h(t) := r_{i_0} + \sum_{n=0}^{N} O_{j_n,i_0}(t_2 - t)$ it holds that $O_{i_0}(t) = (p_{i_0} * h)(t_2 - t)$ where $*$ is the convolution operator.

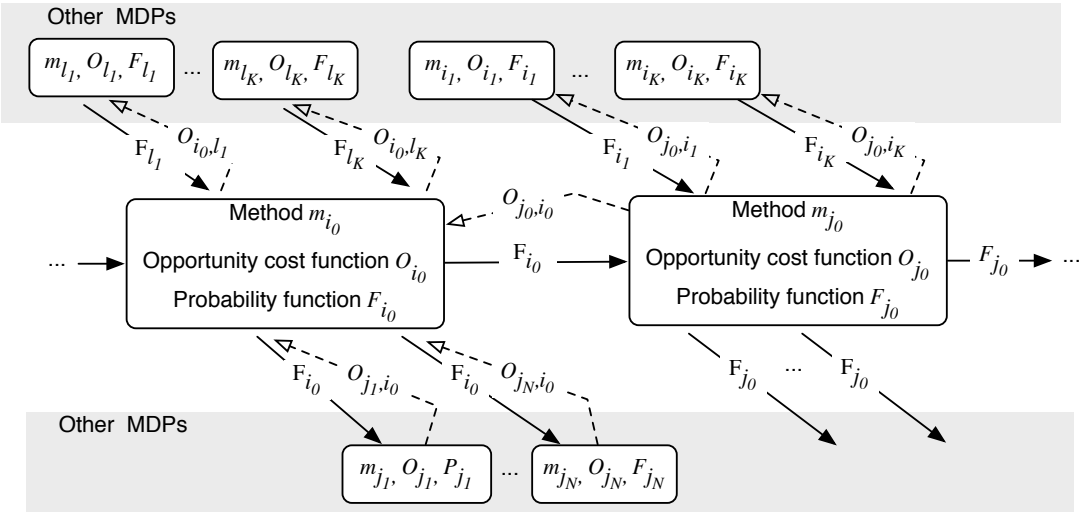

Figure 5.3: Propagation of opportunity cost functions and probability functions through one method: For the opportunity cost function propagation through method m_{i_0}, opportunity cost functions O_{j_n} of methods m_{j_n} for $n = 0, 1, ..., N$ are known, and the opportunity cost function O_{i_0} must be calculated. For the probability function propagation through method m_{j_0}, the probability functions F_{i_k} of methods m_{i_k} for $k = 1, 2, ..., K$ are known, and the probability function F_{j_0} must be calculated.

$O_{j_0,i_0}(t)$ that method m_{i_0} receives for enabling the execution of method m_{j_0} at time t (used in Equation 5.1) is given by:

$$O_{j_0,i_0}(t) = \max_{t' \geq t} \overline{O}_{j_0,i_0}(t') \tag{5.3}$$

Where O_{j_0,i_0} can be greater than $\overline{O}_{j_0,i_0}$ since it can be more profitable to delay the execution of the method m_{i_0} (as illustrated in Figure 5.4).

It has consequently been shown how to propagate the opportunity cost functions: Knowing $\{O_{j_n}\}_{n=0}^{N}$ of methods $\{m_{j_n}\}_{n=0}^{N}$ one can derive O_{i_0} of method m_{i_0} by following Equations 5.2, 5.3 and then 5.1. In general, the opportunity cost function propagation phase starts with sink nodes. It then visits at each time a method m, such that all the methods that m enables have already been

marked as visited. The value function propagation phase terminates when all the source methods have been marked as visited.

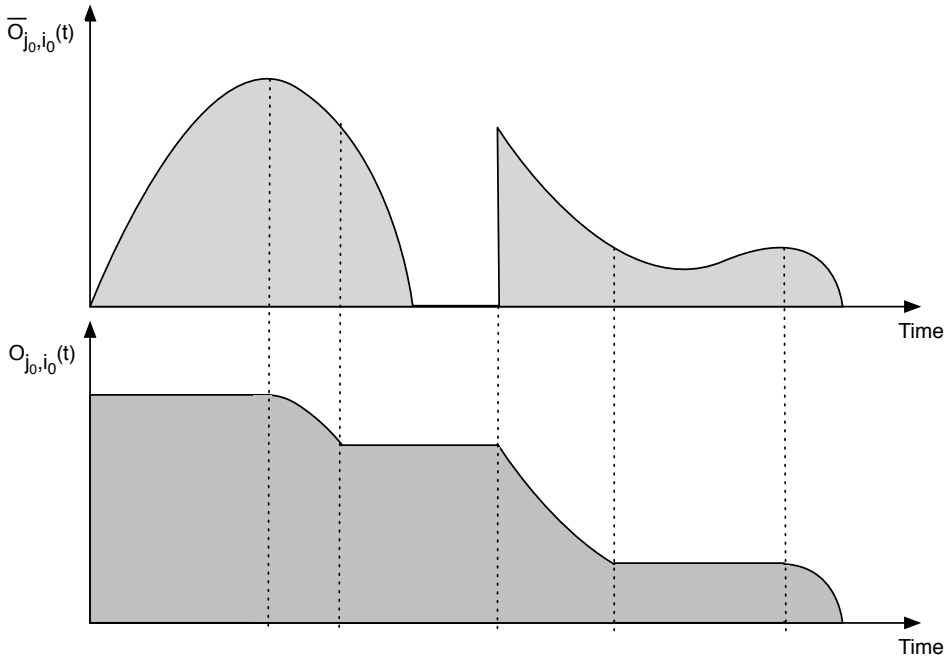

Figure 5.4: Visualization of the operation performed by Equation 5.3

What remains to be shown is how to calculate the value function V_{i_0}, used in Equation 5.1 to identify the policy $\pi_n(\langle i, t\rangle)$. Recall, that the value function V_{i_0} differs from the opportunity cost function O_{i_0} in that the opportunity cost function O_{i_0} assumes that method m_i is enabled (methods m_j such that $\langle i, j\rangle \in C_\prec$ have been executed successfully) whereas the value function V_{i_0} does not make this assumption. When agent n is about to start the execution of method m_{i_0} it knows that its method $m_{l_0} \in M_n$ such that $\langle l_0, i_0\rangle \in C_\prec$ has been successfully executed. However, agent n is still

unsure if other agents have completed their methods $\{m_{l_k}\}_{k=1}^{k=K}$ such that $\langle l_k, i_0 \rangle \in C_{\prec}$. Therefore, the value function V_{i_0} of method m_{i_0} can be calculated as follows:

$$\begin{aligned} V_{i_0}(t) &= O_{i_0}(t) \cdot Pb\left(F_{l_0}(t) = 1 \wedge F_{l_1}(t) = 1 \wedge ... \wedge F_{l_K}(t) = 1\right) \\ &= O_{i_0}(t) \cdot Pb\left(F_{l_1}(t) = 1 \wedge ... \wedge F_{l_K}(t) = 1\right) \\ &= O_{i_0}(t) \prod_{k=1}^{K} F_{l_k}(t). \end{aligned} \tag{5.4}$$

Where the dependency of probability functions $\{F_{l_k}\}_{k=1}^{K}$ has been ignored (similar approximation is used in [Beynier and Mouaddib, 2005], [Beynier and Mouaddib, 2006]).

Finally, in order to determine the policy of agent n for method m_{i_0} one must identify the set Z_{i_0} of intervals of method m_{i_0} activity, that is, intervals $[z, z'] \subset [0, \Delta]$ such that for all $t \in [z, z']$ it holds that $\pi_n(\langle i_0, t \rangle) = E$. As can be seen in Figure 5.1, these intervals of activity in Z_{i_0} are easily identifiable.

5.1.4 Probability Function Propagation Phase

Recall the meaning of the following two functions: $p_i(t)$ is the probability that execution duration of method m_i will consume time t and $F_i(t)$ is the probability that the execution of method m_i will be completed successfully before time t. Assume now that the opportunity cost function propagation phase has been completed and that the opportunity cost functions O_j and sets Z_j for all methods $m_j \in M$ have been calculated. Since the opportunity cost function propagation phase was using probability functions F_i found at previous algorithm iteration (for an old policy), the new probability functions F_i (for the new policy π identified by the current sets Z_j) now have to be found, to prepare the algorithm for the next policy iteration.

The general case of the probability function propagation is shown in Figure 5.3 where the probability functions $\{F_{i_k}\}_{k=0}^{K}$ of methods $\{m_{i_k}\}_{k=0}^{K}$ are known, and the probability function F_{j_0} of method m_{j_0} is to be derived. From the probability function F_{i_0} and the set Z_{j_0} of intervals of activity one can calculate the probability $F'_{j_0}(t)$ that method m_{j_0} will be started before time t:

$$F'_{j_0}(t) = \begin{cases} F_{i_0}(t) & \text{if } \exists [t_1, t_2] \in Z_{j_0} \text{ such that } t \in [t_1, t_2] \\ F_{i_0}(t_2) & \text{otherwise} \end{cases}$$

Where t_2 is such that $[t_1, t_2] \in Z_{j_0}$ is the latest interval of activity before time t. Intuitively, between time t_2 and t the function F'_{j_0} cannot increase because the method will not be started in the time interval $[t_2, t]$. However, for method m_{j_0} to be executed *successfully*, method m_{j_0} has to be enabled, i.e., methods m_{i_1}, ..., m_{i_K} must be successfully finished before method m_{j_0} starts. Hence, Equation 5.5 must be modified, to account for probabilities of enabling method m_{j_0} before time t. Thus, one can calculate the probability $F''_{j_0}(t)$ that method m_{j_0} will be *successfully* started before time t as: (ignoring the dependency of $\{F_{i_k}\}_{k=0}^{K}$)

$$F''_{j_0}(t) = \begin{cases} \prod_{k=0}^{K} F_{i_k}(t) & \text{if } \exists [t_1, t_2] \in Z_{j_0} \text{ such that } t \in [t_1, t_2] \\ \prod_{k=0}^{K} F_{i_k}(t_2) & \text{otherwise} \end{cases} \tag{5.5}$$

Where t_2 is such that $[t_1, t_2] \in Z_{j_0}$ is the latest interval of activity before time t. Probability function F_{j_0} is then derived as follows: For notation convenience, let $f''_{j_0} := \frac{d}{dt} F''_{j_0}$ and $f_{j_0} :=$

$\frac{d}{dt}F_{j_0}$. Also, let $Z_{j_0}(t)$ be a set of time intervals $[z_1, z_2] \in Z_{j_0}$ such that $z_2 \leq t$ augmented with an additional interval $[z_1, t]$ if $z_1 < t < z_2$. It then holds that:

$$
\begin{aligned}
F_{j_0}(t) &= \int_0^t f_{j_0}(y)dy = \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} f_{j_0}(y)dy \\
&\quad \text{Because } f_{j_0}(y) \text{ is equal to 0 outside the intervals of activity of } Z_{j_0} \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} \int_{t_1}^{y} f''_{j_0}(x) \cdot p_{j_0}(y-x)dx\, dy \\
&\quad \text{The execution of method } m_{j_0} \text{ will finish at time } y, \text{ if it starts at time } x \in [t_1, y] \\
&\quad \text{and lasts for } y - x \text{ time units} \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} \int_{t_1}^{y} f''_{j_0}(y-x) \cdot p_{j_0}(x)dy\, dx \\
&\quad \text{Because the arguments } x \text{ and } y - x \text{ in the nested integral are interchangeable} \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} \int_{x}^{t_2} f''_{j_0}(y-x) \cdot p_{j_0}(x)dy\, dx \\
&\quad \text{Order of integration has been reversed (integration ranges are updated accordingly)} \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} p_{j_0}(x) \int_{x}^{t_2} f''_{j_0}(y-x)dy\, dx \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} p_{j_0}(x) \int_{0}^{t_2-x} f''_{j_0}(u)du\, dx \\
&\quad y - x \text{ has been substituted with } u \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} \int_{t_1}^{t_2} p_{j_0}(x) F''_{j_0}(t_2 - x)dx \\
&= \sum_{[t_1,t_2]\in Z_{j_0}(t)} (p_{j_0} * F''_{j_0})(t_2) - (p_{j_0} * F''_{j_0})(t_1) \qquad (5.6)
\end{aligned}
$$

Where F''_{j_0} is given by Equation 5.5.

It has consequently been shown how to propagate the probability functions $\{F_{i_k}\}_{k=0}^{K}$ of methods $\{m_{i_k}\}_{k=0}^{K}$ to obtain the probability function F_{j_0} of method m_{j_0}. The general, the probability

function propagation phase starts with source methods $m_s \in M$ for which F''_s is calculated from Equation 5.5 using known sets Z_s. The algorithm then visits at each time a method m such that all the methods that enable m have already been marked as visited. The probability function propagation phase terminates when all the sink methods have been marked as visited.

To summarize, the VFP algorithm starts with the earliest starting time policy π^0. The algorithm then iterates. At each iteration it: (i) Propagates backwards the opportunity cost functions O_i using the probability functions F_i from the previous algorithm iteration and establishes the new sets Z_i of method activity intervals and then (ii) propagates forwards the new probability functions F_i using the newly established sets Z_i. These new probability functions F_i are then used in the next iteration of the algorithm. Similarly to the OC-DEC-MDP algorithm, the VFP algorithm terminates if a new policy does not improve the policy from the previous algorithm iteration by more than a small value ϵ.

5.1.5 Splitting the Opportunity Cost Functions

In Section 5.1.3 it has been left unresolved how the opportunity cost function O_{j_0} of method m_{j_0} is split into the opportunity cost functions $\{O_{j_0,i_k}\}_{k=0}^{K}$ sent back to methods $\{m_{i_k}\}_{k=0}^{K}$ that enable method m_{j_0}. So far, the approach similar to [Beynier and Mouaddib, 2005] and [Beynier and Mouaddib, 2006] was taken, referred to as the $H_{\langle 1,1 \rangle}$ heuristic. Formally, the $H_{\langle 1,1 \rangle}$ heuristic

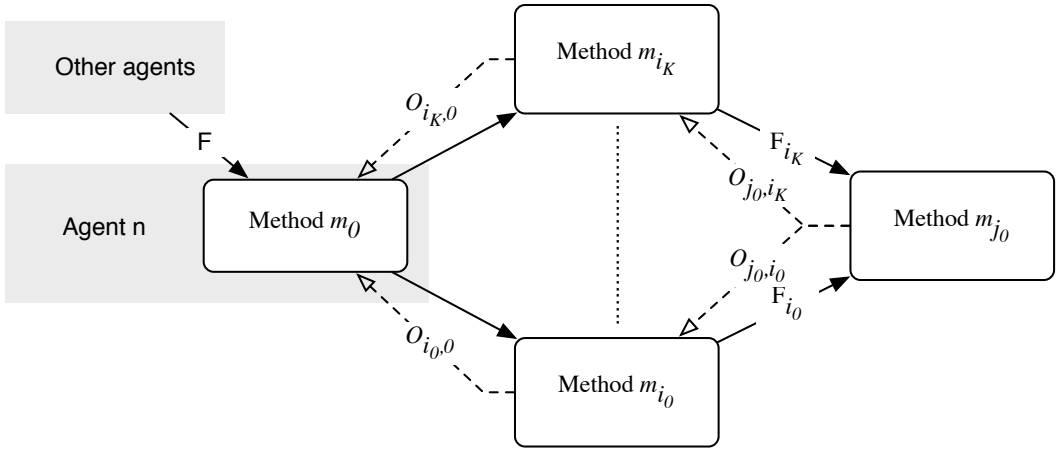

Figure 5.5: Splitting the opportunity cost functions

postulates that, for each method m_{i_k} that enables method m_{j_0}, the opportunity cost $O_{j_0,i_k}(t)$ is given by:

$$\begin{aligned} O_{j_0,i_k}(t) &:= \max_{t'>t} \overline{O}_{j_0,i_k}(t') \\ &= \max_{t'>t}(O_{j_0} \cdot \prod_{\substack{k' \in \{0,\ldots,K\} \\ k' \neq k}} F_{i_{k'}})(t'). \end{aligned} \tag{5.7}$$

It will be proven shortly, that this heuristic overestimates the opportunity cost.

Three problems might arise when splitting the opportunity cost functions: (i) Overestimation, (ii) underestimation or (iii) starvation. Consider the situation in Figure 5.5 where methods $\{m_{i_k}\}_{k=1}^{K}$ enable method m_{j_0}. When the opportunity cost function is propagated through methods $\{m_{i_k}\}_{k=1}^{K}$, for each $k = 0, ..., K$, Equation (5.1) determines the opportunity cost function O_{i_k} from the immediate reward r_k and the opportunity cost function O_{j_0,i_k}. If method m_0 is the only method that enables method m_k, the opportunity cost function $\overline{O}_{i_k,0} = O_{i_k}$ is propagated only to method m_0, and consequently, the opportunity cost for completing the execution of method m_0 at time t is equal to $\sum_{k=0}^{K} O_{i_k,0}(t)$.

If the value of $\sum_{k=0}^{K} O_{i_k,0}(t)$ is overestimated, agent n that is about to start the execution of method m_0 will have too much incentive to finish the execution of method m_0 at time t. Consequently, even when the probability $F(t)$ that method m_0 will be enabled by other agents before time t is low, agent n might still find the expected utility of starting the execution of m_0 at time t higher than the expected utility of doing it later (e.g. in the next time window of method m_0). Hence, at time t the agent will choose to start executing method m_0 rather than waiting, which can have undesirable consequences.

Conversely, if the value of $\sum_{k=0}^{K} O_{i_k,0}(t)$ is underestimated, agent n might loose interest in enabling the execution of future methods $\{m_{i_k}\}_{k=0}^{K}$ and just focus on maximizing the likelihood of earning the immediate reward r_0 for the successful execution of method m_0. Since this likelihood can increase when the agent waits, the agent might decide that at time t it is more profitable to wait rather than to start the execution of method m_0 which can also have undesirable consequences.

Finally, one can easily avoid the overestimation and underestimation of $\sum_{k=0}^{K} O_{i_k,0}(t)$ if the opportunity cost function O_{j_0} is split such that $O_{j_0,i_{k'}} = 1$ and $O_{j_0,i_k} = 0$ for $k \in \{0, 1, ..., K\} \setminus \{k'\}$. However, in such case, agents executing methods m_{i_k} can underestimate the opportunity cost of enabling method m_{j_0} — this problem is referred to as the starvation of method m_k. That short discussion shows the importance of devising heuristics that split the opportunity cost function O_{j_0} in a way that remedies the overestimation, underestimation and starvation problems. In the following it is shown how to arrive at such heuristics.

Theorem 2. *Heuristic $H_{\langle 1,1\rangle}$ overestimates the opportunity cost.*

Proof. To prove the theorem is it sufficient to construct a case where the overestimation occurs. Consider the situation in Figure (5.5) when the $H_{\langle 1,1\rangle}$ heuristic splits the opportunity cost function

O_{j_0} into the opportunity cost functions $\overline{O}_{j_0,i_k} = O_{j_0} \cdot \prod_{\substack{k' \in \{0,...,K\} \\ k' \neq k}} F_{i_{k'}}$ sent to methods m_{i_k} for all $k = 0, 1, ..., K$. Let method rewards be $r_{i_k} = 0$ and method execution time windows be $\langle i_k, 0, \Delta \rangle \in C_{[]}$ for $k = 0, ..., K$. In order to prove that the overestimation of the opportunity cost occurs, time $t_0 \in [0, \Delta]$ must be found for which the opportunity cost $\sum_{k=0}^{K} O_{i_k}(t_0)$ is greater than the opportunity cost $O_{j_0}(t_0)$. For the domain described above, Equation (5.1) states that:

$$O_{i_k}(t) = \int_0^{\Delta - t} p_{i_k}(t') O_{j_0,i_k}(t + t') dt'$$

Summing over all the methods m_{i_k} that enable method m_{j_0} it holds that:

$$\begin{aligned} \sum_{k=0}^{K} O_{i_k}(t) &= \sum_{k=0}^{K} \int_0^{\Delta - t} p_{i_k}(t') O_{j_0,i_k}(t + t') dt' \\ &= \sum_{k=0}^{K} \int_0^{\Delta - t} p_{i_k}(t') \max_{t'' \geq t + t'} \overline{O}_{j_0,i_k}(t + t') dt' \\ &\geq \sum_{k=0}^{K} \int_0^{\Delta - t} p_{i_k}(t') \overline{O}_{j_0,i_k}(t + t') dt' \\ &= \sum_{k=0}^{K} \int_0^{\Delta - t} p_{i_k}(t') O_{j_0}(t + t') \prod_{\substack{k' \in \{0,...,K\} \\ k' \neq k}} F_{i_{k'}}(t + t') dt' \end{aligned} \tag{5.8}$$

Now, let $c \in (0, 1]$ and $t_0 \in [0, \Delta]$ be such that for all $t > t_0$ and $k \in \{0, .., K\}$ it holds that $\prod_{\substack{k' \in \{0,...,K\} \\ k' \neq k}} F_{i_{k'}}(t) > c$. The lower bound of $\sum_{k=0}^{K} O_{i_k}(t_0)$ for the argument t_0 is then as follows:

$$\sum_{k=0}^{K} O_{i_k}(t_0) > \sum_{k=0}^{K} \int_0^{\Delta - t_0} p_{i_k}(t') O_{j_0}(t_0 + t') \cdot c \ dt'$$

Because $F_{j_{k'}}$ is non-decreasing. Now, suppose that there exists a time point $t_1 \in (t_0, \Delta]$, such that $\sum_{k=0}^{K} \int_0^{t_1 - t_0} p_{i_k}(t') dt' > O_{j_0}(t_0)/(c \cdot O_{j_0}(t_1))$. Since the decrease of the upper limit of the integration

from $\Delta - t_0$ to $t_1 - t_0$ decreases the value of the integral (function under the integral is positive) it holds that:

$$\sum_{k=0}^{K} O_{i_k}(t_0) \quad > \quad c \sum_{k=0}^{K} \int_0^{t_1 - t_0} p_{i_k}(t') O_{j_0}(t' + t_0) dt'$$

And since $O_{j_0}(t' + t_0)$ is non-increasing:

$$\sum_{k=0}^{K} O_{i_k}(t_0) \quad > \quad c \cdot O_{j_0}(t_1) \sum_{k=0}^{K} \int_0^{t_1 - t_0} p_{i_k}(t') dt'$$
$$\sum_{k=0}^{K} O_{i_k}(t_0) \quad > \quad c \cdot O_{j_0}(t_1) \frac{O_j(t_0)}{c \cdot O_j(t_1)} = O_j(t_0)$$

Consequently, the opportunity cost $\sum_{k=0}^{K} O_{i_k}(t_0)$ for starting the execution of methods $\{m_{i_k}\}_{k=0}^{K}$ at time $t \in [0, \Delta]$ is greater than the opportunity cost $O_{j_0}(t_0)$ which proves the theorem. The overestimation of the opportunity cost caused by using the $H_{\langle 1,1 \rangle}$ heuristic is visualized in Figure 6.1.1. □

In the following, three heuristics that remedy the opportunity cost overestimation problem are proposed:

- **Heuristic** $H_{\langle 1,0 \rangle}$: Only method m_{i_k} gets the reward for enabling method m_{j_0}, i.e., $\overline{O}_{j_0,i_k}(t) = (O_{j_0} \cdot \prod_{\substack{k' \in \{0,...,K\} \\ k' \neq k}} F_{i_{k'}})(t)$ and $\overline{O}_{j_0,i_{k'}}(t) = 0$ for $k' \in \{0, ..., K\} \backslash \{k\}$.
- **Heuristic** $H_{\langle 1/2,1/2 \rangle}$: Each method m_{i_k} for $k = 0, 1, ..., K$ gets the full opportunity cost for enabling method m_{j_0} divided by the number K of methods enabling method m_{j_0}, i.e., $\overline{O}_{j_0,i_k}(t) = \frac{1}{K}(O_{j_0} \cdot \prod_{\substack{k' \in \{0,...,K\} \\ k' \neq k}} F_{i_{k'}})(t)$ for $k \in \{0, ..., K\}$.

- **Heuristic** $\widehat{H}_{\langle 1,1\rangle}$: This is a normalized version of the $H_{\langle 1,1\rangle}$ heuristic in that each method m_{i_k} for $k = 0, 1, ...K$ *initially* (for small t) gets the full opportunity cost for enabling method m_{j_0}. In order to prevent the opportunity cost overestimation the functions $\{O_{j_0,i_k}\}_{k=0}^{K}$ are normalized when their sum exceeds the opportunity cost function to be split. Formally:

$$\overline{O}_{j_0,i_k}(t) = \begin{cases} O_{j_0,i_k}^{H_{\langle 1,1\rangle}}(t) & \text{if } \sum_{k=0}^{K} O_{j_0,i_k}^{H_{\langle 1,1\rangle}}(t) < O_{j_0}(t) \\ O_{j_0}(t)\frac{O_{j_0,i_k}^{H_{\langle 1,1\rangle}}(t)}{\sum_{k=0}^{K} O_{j_0,i_k}^{H_{\langle 1,1\rangle}}(t)} & \text{otherwise} \end{cases}$$

Where $O_{j_0,i_k}^{H_{\langle 1,1\rangle}}(t) := (O_{j_0} \cdot \prod_{\substack{k'\in\{0,...,K\}\\ k'\neq k}} F_{j_{k'}})(t)$.

For the new heuristics it then holds that:

Theorem 3. *Heuristics $H_{\langle 1,0\rangle}$, $H_{\langle 1/2,1/2\rangle}$ and $\widehat{H}_{\langle 1,1\rangle}$ do not overestimate the opportunity cost.*

Proof. When the $H_{\langle 1,0\rangle}$ heuristic is used to split the opportunity cost function O_{j_0} only one method (e.g. m_{i_k}) gets the opportunity cost for enabling method m_{j_0}. Thus:

$$\begin{aligned} \sum_{k'=0}^{K} O_{i_{k'}}(t) &= \int_0^{\Delta-t} p_{i_k}(t')O_{j_0,i_k}(t+t')dt' \qquad (5.9) \\ &\quad \text{And since } O_{j_0} \text{ is non-increasing} \\ &\leq \int_0^{\Delta-t} p_{i_k}(t')O_{j_0}(t+t') \cdot \prod_{\substack{k'\in\{0,...,K\}\\ k'\neq k}} F_{j_{k'}}(t+t')dt' \\ &\leq \int_0^{\Delta-t} p_{i_k}(t')O_{j_0}(t+t')dt' \leq O_{j_0}(t) \end{aligned}$$

The last inequality holds because the opportunity cost function O_{j_0} is non-increasing.

For the $H_{\langle 1/2,1/2\rangle}$ heuristic the proof is similar:

$$\begin{aligned}\sum_{k=0}^{K} O_{i_k}(t) &\leq \sum_{k=0}^{K} \int_0^{\Delta-t} p_{i_k}(t') \frac{1}{K} O_{j_0}(t+t') \prod_{\substack{k' \in \{0,\ldots,K\} \\ k' \neq k}} F_{j_{k'}}(t+t') dt' \\ &\leq \frac{1}{K} \sum_{k=0}^{K} \int_0^{\Delta-t} p_{i_k}(t') O_{j_0}(t+t') dt' \\ &\leq \frac{1}{K} \cdot K \cdot O_{j_0}(t) = O_{j_0}(t).\end{aligned}$$

For the $\widehat{H}_{\langle 1,1\rangle}$ heuristic the opportunity cost function O_{j_0} is by definition split such that $\sum_{k=0}^{K} O_{i_k}(t) \leq O_{j_0}(t)$. Thus, is has been proven that the new heuristics $H_{\langle 1,0\rangle}$, $H_{\langle 1/2,1/2\rangle}$ and $\widehat{H}_{\langle 1,1\rangle}$ prevent the overestimation of the opportunity cost. □

The reason why all three heuristics have been introduced is the following: Since the $H_{\langle 1,1\rangle}$ heuristic overestimates the opportunity cost, one has to choose which method m_{i_k} should receive the reward for enabling the method m_{j_0} which is exactly what the $H_{\langle 1,0\rangle}$ heuristic postulates. However, the $H_{\langle 1,0\rangle}$ heuristic does not reward the other methods for enabling method m_{j_0} which introduces the starvation problem. Starvation can be avoided if the opportunity cost functions are split using the $H_{\langle 1/2,1/2\rangle}$ heuristic that provides reward to all enabling methods. However, the sum of the split opportunity cost functions for the $H_{\langle 1/2,1/2\rangle}$ heuristic can be smaller than the non-zero opportunity cost function for the $H_{\langle 1,0\rangle}$ heuristic, which is clearly undesirable. This is why the $\widehat{H}_{\langle 1,1\rangle}$ heuristic which by definition avoids the overestimation, underestimation and starvation problems has been introduced.

5.1.6 Implementation of Function Operations

In the previous sections all the function derivations have been carried out without choosing a particular representation of the underlying continuous functions. In general, one can choose from various function approximation techniques such as the piecewise linear [Boyan and Littman, 2000], piecewise constant [Li and Littman, 2005], or piecewise gamma [Marecki et al., 2007] approximation. In the actual implementation of VFP evaluated in Section 6.1, piecewise linear function representation has been chosen.

When VFP propagates the opportunity cost functions and probability functions it carries out operations represented by Equations (5.1) and (5.6) which are equivalent to computing the convolution function $f(t) = (p * h)(t)$. If time is discretized, functions $p(t)$ and $h(t)$ are discrete; however, $h(t)$ can be approximated with a piecewise linear function $\widehat{h}(t)$, which is exactly what the VFP algorithm does. As a result, instead of performing $O(\Delta^2)$ multiplications to compute $f(t)$ (as in case of the OC-DEC-MDP algorithm), VFP only needs to perform $O(k \cdot \Delta)$ multiplications to compute $f(t)$ where k is the number of linear segments of $\widehat{h}(t)$ and $k < \Delta$. Since the values of F_i are in range $[0, 1]$ and the values of O_i are in range $[0, \sum_{m_i \in M} r_i]$, opportunity cost functions O_i are chosen to be approximated within the error ϵ_O and the probability functions F_i are chosen to be approximated within the error ϵ_F. The experimental results in Section 6.1 demonstrate the tradeoff between the runtime and solution quality of VFP for different values of ϵ_O and ϵ_F.

5.2 Globally Optimal Solution: The M-DPFP Algorithm

This section introduces an algorithm (M-DPFP[3]) that finds solutions to CR-DEC-MDPs whose quality is guaranteed to be within an arbitrary small ϵ of the quality of the optimal solution. The idea of M-DPFP is the following: Recall the definition of the CR-DEC-MDP model from Section 2.2.3 where $\mathcal{S}_n$ is the set of states of agent n. Determining the optimal actions for all states $s \in \mathcal{S}_n$ is impossible since each state s is specified by continuous variables (method starting and finishing times) and consequently, set $\mathcal{S}_n$ contains an infinite number of elements. Yet, it is possible to leverage the DPFP algorithm from Section 3.2 to find ϵ-optimal actions for a finite number of states in $\mathcal{S}_n$ — states referred to as the **representative states**. One can then prescribe a greedy policy to non-representative states such that the error of M-DPFP is still expressible in terms of an arbitrary small parameter ϵ. These ideas are first shown on an example (Section 5.2.1) and later generalized to an arbitrary CR-DEC-MDP (Sections 5.2.2).

5.2.1 Arriving at M-DPFP

Consider the search for an optimal policy for the planning problem introduced in Section 2.1.3.2. Here, both agents know exactly the starting states $s_{1,0} \in \mathcal{S}_1$ and $s_{2,0} \in \mathcal{S}_2$. However, since no explicit communication is allowed after the planning phase, each agent can only estimate the current state of the other agent during the execution phase. For example, if agent 2 finishes the execution of its method m_4 successfully it can infer that agent 1 has successfully finished the execution of its method m_1 (see Figures 5.2.1 and 5.6).

[3]The M-DPFP algorithm exploits the idea to perform the forward search in the space of cumulative distribution functions introduced by the DPFP algorithm from Section 3.2, hence the name M-DPFP.

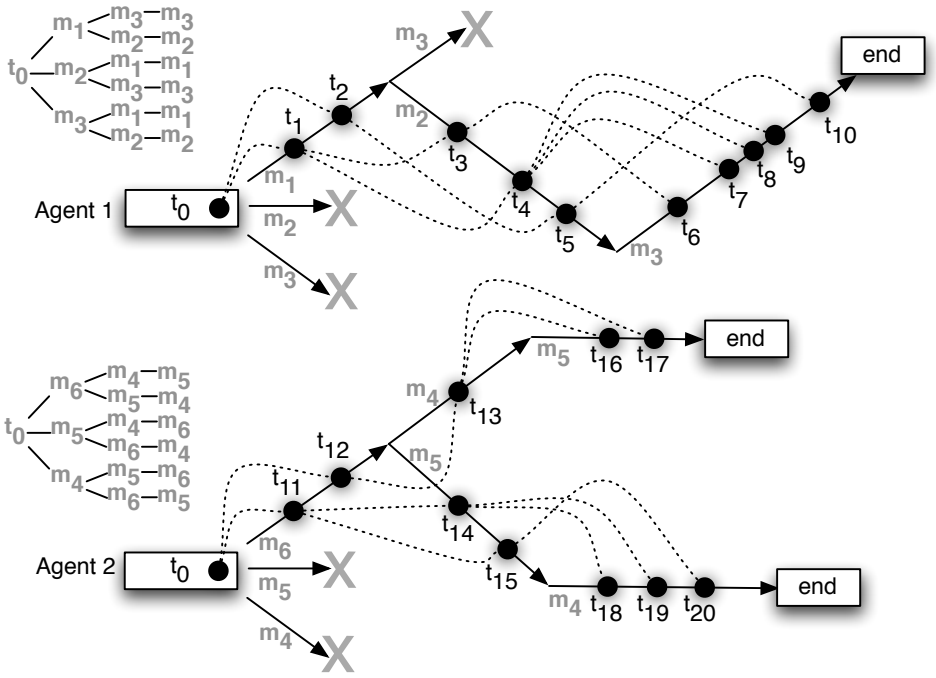

Figure 5.6: M-DPFP algorithm: Generation of the representative states. Solid arrows represent time axes whereas dotted lines passing through representative states illustrate example execution histories. Each time point t_i has a corresponding state s_i (see Figure 5.2.1).

Adding Representative States

At a basic level, the M-DPFP algorithm intertwines the generation of representative states with the search for the ϵ-optimal actions to be executed from these states. When the algorithm starts, it generates representative states $s_{1,0} = (\langle -1, t_0, t_0, 1\rangle)$ and $s_{2,0} = (\langle -2, t_0, t_0, 1\rangle)$ which are simply the starting states of agents 1 and 2 (see the description of states in the CR-DEC-MDP model in Section 2.2.3). The algorithm then searches for the best actions to be executed from $s_{1,0}$ and $s_{2,0}$ using a fast **lookahead function** (similar to the DPFP algorithm from Section 3.2).

Suppose that the lookahead function found that the best action to be executed from state $s_{1,0}$ is $m_1 \in A(s_{1,0})$ and the best action to be executed from state $s_{2,0}$ is $m_6 \in A(s_{2,0})$ (as illustrated

$$
\begin{aligned}
s_{1,0} &= (\langle -1, t_0, t_0, 1\rangle)\\
s_1 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle)\\
s_3 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_3, 1\rangle)\\
s_6 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_3, 1\rangle, \langle 3, t_3, t_6, 1\rangle)\\
s_4 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_4, 1\rangle)\\
s_7 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_4, 1\rangle, \langle 3, t_4, t_7, 1\rangle)\\
s_8 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_4, 1\rangle, \langle 3, t_4, t_8, 1\rangle)\\
s_9 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_4, 1\rangle, \langle 3, t_4, t_9, 1\rangle)\\
s_2 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_2, 1\rangle)\\
s_5 &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_2, 1\rangle, \langle 2, t_2, t_5, 1\rangle)\\
s_{10} &= (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_2, 1\rangle, \langle 2, t_2, t_5, 1\rangle, \langle 3, t_5, t_{10}, 1\rangle)\\
\\
s_{2,0} &= (\langle -2, t_0, t_0, 1\rangle)\\
s_{11} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle)\\
s_{14} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle, \langle 5, t_{11}, t_{14}, 1\rangle)\\
s_{18} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle, \langle 5, t_{11}, t_{14}, 1\rangle, \langle 4, t_{14}, t_{18}, 1\rangle)\\
s_{19} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle, \langle 5, t_{11}, t_{14}, 1\rangle, \langle 4, t_{14}, t_{19}, 1\rangle)\\
s_{15} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle, \langle 5, t_{11}, t_{15}, 1\rangle)\\
s_{20} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{11}, 1\rangle, \langle 5, t_{11}, t_{15}, 1\rangle, \langle 4, t_{15}, t_{20}, 1\rangle)\\
s_{12} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{12}, 1\rangle)\\
s_{13} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{12}, 1\rangle, \langle 4, t_{12}, t_{13}, 1\rangle)\\
s_{16} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{12}, 1\rangle, \langle 4, t_{12}, t_{13}, 1\rangle, \langle 5, t_{13}, t_{16}, 1\rangle)\\
s_{17} &= (\langle -2, t_0, t_0, 1\rangle, \langle 6, t_0, t_{12}, 1\rangle, \langle 4, t_{12}, t_{13}, 1\rangle, \langle 5, t_{13}, t_{17}, 1\rangle)
\end{aligned}
\tag{5.10}
$$

Figure 5.7: Representative states for the search problem from Figure 5.6. Note that only the representative states for the successful execution of methods are shown.

in Figure 5.6). The algorithm now switches to generating new representative states. For agent 1, the execution of method m_1 can finish at any point in time (inside the execution time window of method m_1), yet, the algorithm can only consider a finite set of representative states that agent 1 can transition to having completing the execution of method m_1. Figure 5.6 illustrates a situation when the algorithm considers just two representative states: $s_1 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle)$ and

$s_2 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_2, 1\rangle)$. In this particular case both s_1 and s_2 correspond to a situation where agent 1 has executed method m_1 successfully (with $q = 1$)[4], yet in general, both successful and unsuccessful cases are considered. Furthermore, s_1 corresponds to a situation where the execution of method m_1 finished at time t_1 and s_2 corresponds to a situation where the execution of method m_1 finished at time t_2. While it is explained later how the algorithm selects these finishing times, assume for now that they are arbitrary.

The algorithm then switches back to the search for the ϵ-optimal actions to be executed from states s_1 and s_2 (using the lookahead function) which turn out to be the same, i.e., to execute method m_2 (as illustrated in Figure 5.6). Next, the algorithm switches back to the generation of new representative states, for a situation where the execution of method m_2 starts in state s_1 as well as for a situation where the execution of method m_2 starts in state s_2. In the example from Figure 5.6, the algorithm has chosen to generate two new representative states $s_3 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_3, 1\rangle)$ and $s_4 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_4, 1\rangle)$ for the execution starting in s_1 and just one new representative state $s_5 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_2, 1\rangle, \langle 2, t_2, t_5, 1\rangle)$ for the execution starting in s_2. The algorithm then continues to intertwine the generation of representative states with the search for the ϵ-optimal actions to be executed from these states using the technique just discussed. Note, that this approach allows M-DPFP to prune many possible policies. For example, in Figure 5.6 agent 1 will never choose to execute method m_3 after finishing the execution of method m_1.

[4]Recall that states are sequences of events $e = \langle i, l_1, l_2, q\rangle$ where q is the is the result of the execution of method m_i

Using the Lookahead Function

The lookahead function mentioned above (explained in detail in Section 5.2.4) not only finds the ϵ-optimal action to be executed from a representative state but also returns the probability distribution of entering future states (after that representative state) when following the ϵ-optimal policy from that representative state. For example, when the lookahead function is called to find the ϵ-optimal action to be executed from a representative state $s_1 = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle)$ it also returns the probability of entering future states $s = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, ...)$. Note however, that these probabilities are not only affected by agent 1's actions, but also by agent 2's actions, e.g., the probability of transitioning to state $s = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_3, 1\rangle)$ is affected by the probability that agent 2 will successfully complete the execution of its method m_5 before time t_3. Consequently, for the the lookahead function to correctly determine the probabilities of entering future states $s = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, ...)$ it must know agent 2's policy first.

To this end it is required that the lookahead function considers the representative states in a specific order explained in depth in Section 5.2.2. In essence, this specific order ensures that when the lookahead function is called for a representative state s, the lookahead function can correctly estimate (given the information provided in the description of state s) the probability distribution over the current states of all the agents which may have an impact on the policy executed from s. For example, in order to estimate the expected utility of starting the execution of method m_2 of agent 1 from state s_1 (at time t_1) we need to know the probability with which method m_5 of agent 2 may have finished before time t_1 (because method m_5 enables method m_2). In the example in Figure 5.6 the ϵ-optimal action for agent 2 in state $s_{2,0}$ is to choose to execute method m_6, not

m_5. So, considering only the ϵ-optimal action from state $s_{2,0}$ is not enough as it still does not tell us the probability of method m_5's ending before time t_1. (This does not yet provide us enough information to compute expected utility of starting to execute method m_2 from state s_1, because we still cant tell the likelihood of finishing the execution of method m_5 before time t_1). So now we must consider the ϵ-optimal actions that agent 2 will execute after completing the execution of method m_6, in particular, the ϵ-optimal actions from the representative states s_{11} and s_{12} must be found. For state s_{12} the ϵ-optimal action is still not to execute m_5 and we must continue to consider state s_{13} before the likelihood of finishing the execution of method m_5 before time t_1 can be estimated.

To further illustrate the specific ordering in which the lookahead function can be called consider the following two examples (refer to Figure 5.6). Initially, the lookahead function can be called for the starting state $s_{1,0}$ or the starting state $s_{2,0}$ in any order because either way, the lookahead function knows that the probability distribution over the current states of agent 1 is $Pb(s_{1,0} = 1)$ and that the probability distribution over the states of agent 2 is $Pb(s_{2,0} = 1)$. On the other hand, having been called *only* for the representative state $s_{1,0}$, the lookahead function cannot be immediately called for the representative state s_1 since at this point the probability distribution over the current states of agent 2 is unknown (because the lookahead function has not been called for state $s_{2,0}$ yet). In the next section the specific ordering in which the lookahead function considers the representative states is formalized.

5.2.2 The M-DPFP Algorithm

In order to formalize the M-DPFP algorithm it is necessary properly define the ordering in which M-DPFP adds representative states, that is, define the concept of a method sequence and a method

sequences graph. The formal definition of these two concepts is followed by a concrete example (refer to Figure 5.8).

Definition 5. Method sequence *ϕ of agent n is a vector $\phi = (m_{-n}, m_{i_1}, ..., m_{i_k})$ of executed methods where m_{-n} is a spoof method completed by agent n before the execution starts and $m_{i_1}, ..., m_{i_k} \in M_n$. Furthermore, Φ_n is the set of all possible sequences ϕ of agent n and $\Phi = \bigcup_{n=1}^{N} \Phi_n$ is the set of all possible method sequences of all agents. Also, for notational convenience, (ϕ, m) denotes a method sequence that is a concatenation of vector ϕ with vector (m) whereas $(..., m)$ denotes a method sequence that ends with a method m. Finally, the set S_ϕ of* ***states associated with method sequence*** *ϕ is defined as:*

$$S_\phi = \{(\langle i_1, l_{i_1,1}, l_{i_1,2}, q_{i_1}\rangle, \ldots, \langle i_k, l_{i_k,1}, l_{i_k,2}, q_{i_k}\rangle) \in S \text{ such that } (m_{i_1}, ..., m_{i_k}) = \phi\}.$$

For example, in Figure 5.8 method sequences of agent 1 can be (m_{-1}, m_1), (m_{-1}, m_1, m_2), (m_{-1}, m_1, m_3), etc. As mentioned in Section 5.2.1, in order to take advantage of the lookahead function, the representative states must be added in a specific order, i.e., the lookahead function must be able to estimate the probability distribution over the current states of all the agents (given the information provided in the description of the current representative state). To this end, M-DPFP adds representative states one method sequence at a time, considering subsequent method sequences from a list $\mathcal{L}$ defined as follows:

Definition 6. Method sequences graph *is a directed graph $G = (\Phi, C)$ where Φ is the set of nodes and C is the set of arcs. The set of arcs C is constructed as follows: For any two nodes $\phi_i = (..., m_i), \phi_j = (..., m_j) \in \Phi$ an arc (ϕ_i, ϕ_j) is added to C if $\phi_j = (\phi_i, m)$ or $\langle i, j\rangle \in C_\prec$.*

The list $\mathcal{L}$ of method sequences is then a topological sorting of the elements of Φ, that is, a method sequence ϕ_j is added at the end of list $\mathcal{L}$ only if all the nodes ϕ_i that precede node ϕ_j in

graph G (all the nodes ϕ_i such that there exists an arc $(\phi_i, \phi_j) \in C$) have already been added to list $\mathcal{L}$.

To illustrate the concept of method sequences, method sequences graph and list $\mathcal{L}$ consider the example in Figure 5.8. Here, each node label is a method sequence, e.g. node -2564 corresponds to method sequence (m_{-2}, m_5, m_6, m_4) of agent 2. Also, arcs represents precedence constraints. For example, node -2564 (that represents the execution of method m_4 after the execution of methods m_5 and m_6) is preceded by node -256 (that represents the execution of method m_6 after the execution of methods m_4). Furthermore node -2564 is preceded by nodes -121, -1231, -11, -131, -1321 because $\langle 4, 1\rangle \in C_{\prec}$ (the execution of method m_4 must be preceded by the successful execution of method m_1). The list $\mathcal{L}$ for the graph in Figure 5.8 constructed from the topological sorting of the the graph nodes can therefore be: $\mathcal{L} = ((m_{-1}),$ $(m_{-2}),$ $(m_{-1}, m_1),$ $(m_{-1}, m_1, m_3),$ $(m_{-1}, m_3),$ $(m_{-1}, m_3, m_1),$ $(m_{-2}, m_5),$ $(m_{-2}, m_5, m_6),$ $(m_{-2}, m_6),$ $(m_{-2}, m_6, m_5),$ $(m_{-1}, m_2),$ $(m_{-1}, m_2, m_1),$ $(m_{-1}, m_2, m_1, m_3),$ $(m_{-1}, m_2, m_3),$ $(m_{-1}, m_2, m_3, m_1),$ $(m_{-2},$ $m_4),$ $(m_{-2}, m_4, m_6),$ $(m_{-2}, m_4, m_6, m_5),$ $(m_{-2}, m_4, m_5),$ $(m_{-2}, m_4, m_5, m_6),$ $(m_{-1}, m_3, m_2),$ $(m_{-1}, m_3,$ $m_2, m_1),$ $(m_{-2}, m_5, m_4),$ $(m_{-2}, m_5, m_4, m_6),$ $(m_{-2}, m_5, m_6, m_4),$ $(m_{-2}, m_6, m_4),$ $(m_{-2}, m_6, m_4, m_5),$ $(m_{-1},$ $m_3, m_1, m_2),$ $(m_{-1}, m_1, m_2),$ $(m_{-1}, m_1, m_2, m_3),$ $(m_{-1}, m_1, m_3, m_2),$ $(m_{-2}, m_6, m_5, m_4))$.

At a basic level, the M-DPFP algorithm iterates over all the method sequences ϕ from $\mathcal{L}$. In particular, for a method sequence ϕ, it performs the following three steps:

- Constructs a finite set $\widetilde{S}_\phi \subset S_\phi$ of representative states.
- For each representative state $s \in \widetilde{S}_\phi$ calls the lookahead function to determine the ϵ-optimal action to be executed from s.

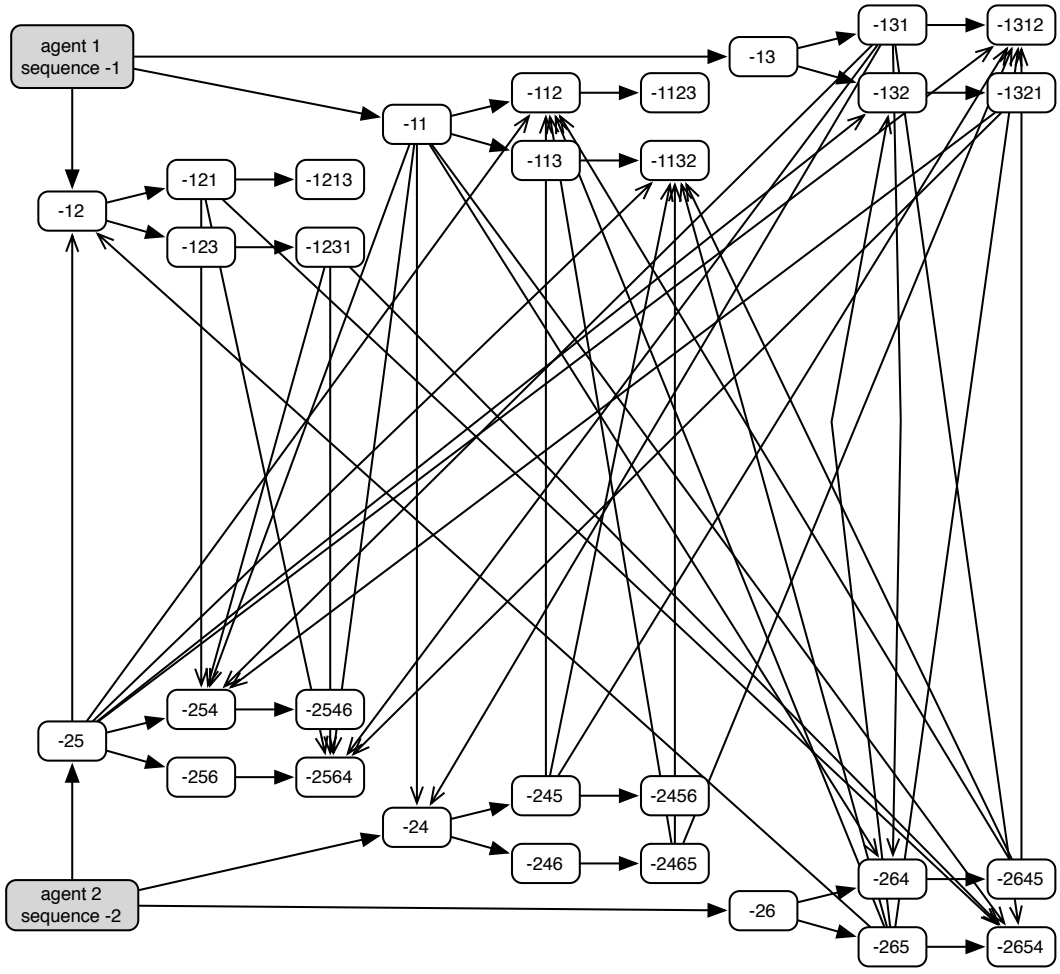

Figure 5.8: Method sequences graph

- Prunes from the list $\mathcal{L}$ all the method sequences that cannot be encountered while executing the ϵ-optimal policy.

Precisely, the algorithm starts by calling the ArrangeSequences(Φ) function to arrange the method sequences into a list $\mathcal{L}$ (topological sort of method sequences in Φ). The algorithm then calls the lookahead function OptimalMethod($s_{i,0}$) for the starting state $s_{i,0}$ over all the agents in order to find the ϵ-optimal actions to be executed from these starting states as well as to prune from list $\mathcal{L}$ some of the method sequences that are not reachable via the optimal policy. The algorithm then iterates over the remaining method sequences in $\mathcal{L}$. In particular, for a sequence

$\phi \in \mathcal{L}$ it first identifies an agent i that this sequence belongs to. The algorithm then incrementally builds a set $\widetilde{S}_\phi$ of representative states and a set Z_ϕ of optimal actions in these states. Initially, both $\widetilde{S}_\phi$ and Z_ϕ are empty.

If the error computeError($\widetilde{S}_\phi$) of the current policy for all states $s \in S_\phi$ is greater than the tolerable error *maxError* the algorithm tries to reduce the error ComputeError($\widetilde{S}_\phi$) by expanding the set $\widetilde{S}_\phi$ of representative states with a new representative state $s \in S_\phi$ (functions NewRepresentativeState($S_\phi, \widetilde{S}_\phi$) and ComputeError($\widetilde{S}_\phi$) are described in Section 5.2.3). To this end, the algorithm calls the lookahead function OptimalMethod(s) to find the optimal method m to be executed from state s of agent i (described in Section 5.2.4), assigns action m to the optimal policy $\pi^*(s)$ for state s and updates sets Z_ϕ and $\widetilde{S}_\phi$ respectively.

As soon as the error ComputeError($\widetilde{S}_\phi$) is within the tolerable error *maxError* the algorithm stops expanding the set $\widetilde{S}_\phi$ with new representative states. It then employs the set Z_ϕ to remove from the list $\mathcal{L}$ the method sequences that are not reachable via the optimal policy. The M-DPFP algorithm terminates if all method sequences from the pruned list $\mathcal{L}$ have been analyzed. At this point, the algorithm returns the optimal policy π^* defined for representative states. The error of the greedy policy for the non-representative states (described in Section 5.2.3.3) is then guaranteed to be less than *maxError*.

In the following three sections it is explained how to:

- Select new representative states (the NewRepresentativeState($S_\phi, \widetilde{S}_\phi$) function);
- Bound the error of the algorithm (the computeError($\widetilde{S}_\phi$) function);
- Find the optimal action to be executed from a representative state (the OptimalAction(s) function).

Algorithm 2 M-DPFP(*maxError*)

$\mathcal{L} \leftarrow$ ArrangeSequences(Φ)
for all Agents $i = 1, 2, ..., N$ **do**
  $\pi^*(s_{i,0}) \leftarrow$ OptimalAction($s_{i,0}$)
  $\mathcal{L} \leftarrow \mathcal{L} \setminus \{(m_{-i}, m, ...) \in \Phi_i \text{ such that } m \neq \pi^*(s_{i,0})\}$
for all $\phi \in \mathcal{L}$ **do**
  $i \leftarrow n$ such that $\phi \in \Phi_n$
  $Z_\phi \leftarrow \emptyset$
  $\widetilde{S}_\phi \leftarrow \emptyset$
  while $maxError >$ ComputeError($\widetilde{S}_\phi$) **do**
    $s \leftarrow$ NewRepresentativeState($S_\phi, \widetilde{S}_\phi$)
    $m \leftarrow$ OptimalAction(s)
    $\pi^*(s) \leftarrow m$
    $Z_\phi \leftarrow Z_\phi \cup m$
    $\widetilde{S}_\phi \leftarrow \widetilde{S}_\phi \cup s$
  $\mathcal{L} \leftarrow \mathcal{L} \setminus \{(\phi, m', ...) \in \Phi_i \text{ such that } m' \notin Z_\phi\}$
return π^*

5.2.3 Representative States

Two heuristics according to which new representative states are selected are introduced in this section. Then, the greedy policy for non-representative states is explained. Finally, the formula for calculating the error of the M-DPFP algorithm is provided.

5.2.3.1 Representative State Selection

Consider the situation when the M-DPFP algorithm calls the NewRepresentativeState($S_\phi, \widetilde{S}_\phi$) function to pick a new representative state s (and later added to the set $\widetilde{S}_\phi$ of representative states associated with the method sequence $\phi \in \mathcal{L}$). Suppose that $\phi = (\phi_{-1}, m_k)$ for some method sequence $\phi_{-1} \in \mathcal{L}$ and method $m_k \in M$. For example, in Figure 5.6, $\phi = (m_{-1}, m_1, m_2)$, $\phi_{-1} = (m_{-1}, m_1)$ and $m_k = m_2$.

Upon considering a method sequence ϕ, the M-DPFP algorithm has already considered the method sequence ϕ_{-1} because by definition, ϕ_{-1} is before ϕ in list $\mathcal{L}$. Furthermore, there exists at least one representative state $s_{-1} \in \widetilde{S}_{\phi_{-1}}$ for which the optimal action is to start executing method m_k because otherwise, ϕ would have been pruned from $\mathcal{L}$ (it would have not been visitable via the optimal policy). For each such representative state s_{-1} the execution of method m_k can finish with an outcome $q \in \{0, 1\}$. Hence, the set $\widetilde{S}_\phi \subset S_\phi$ of representative states for sequence ϕ is given by:

$$\widetilde{S}_\phi = \bigcup_{\substack{s_{-1} \in \widetilde{S}_{\phi_{-1}} \\ \pi(s_{-1}) = m_k \\ q \in \{0,1\}}} \widetilde{S}_{\phi, s_{-1}, q}$$

Where $\widetilde{S}_{\phi,s_{-1},q}$ is the set of representative states that the agent can transition to after it finished executing method m_k from state s_{-1} with an outcome q. (The value $|\widetilde{S}_\phi|/|\widetilde{S}_{\phi_{-1}}|$ is referred to as the *branching factor* for representative states.)

For example, in Figure 5.6 where only successful execution of methods have been considered (for explanatory purposes) we have $s_{-1} \in \widetilde{S}_{(m_{-1},m_1)} = \{s_1, s_2\}$ and hence:

$$\begin{aligned} \widetilde{S}_{(m_{-1},m_1,m_2)} &= \widetilde{S}_{(m_{-1},m_1,m_2),s_1,0} \cup \widetilde{S}_{(m_{-1},m_1,m_2),s_1,1} \cup \widetilde{S}_{(m_{-1},m_1,m_2),s_2,0} \cup \widetilde{S}_{(m_{-1},m_1,m_2),s_2,1} \\ &= \{s_3, s_4\} \cup \{s_5\} = \{s_3, s_4, s_5\}. \end{aligned}$$

because sets $\widetilde{S}_{(m_{-1},m_1,m_2),s_1,0}$ and $\widetilde{S}_{(m_{-1},m_1,m_2),s_2,0}$ are empty (for explanatory purposes).

When the NewRepresentativeState($S_\phi, \widetilde{S}_\phi$) function is called, it first selects a state $s_{-1} \in \widetilde{S}_{\phi_{-1}}$ and an outcome $q \in \{0, 1\}$. (The selection mechanism considers all possible combinations of $s_{-1} \in \widetilde{S}_{\phi_{-1}}$ and $q \in \{0, 1\}$). The NewRepresentativeState function then attempts to pick a new representative state $s \in S_\phi$ and add it to the set $\widetilde{S}_{\phi,s_{-1},q}$. Let $s_{-1} = (e_1, ..., e_m)$ and recall that

method m_k has been executed in state s_{-1} with an outcome q. Initially, when set $\widetilde{S}_{\phi,s_{-1},q} = \emptyset$, the NewRepresentativeState function adds two delimiting representative states to it: The first representative state is associated with the earliest completion time of method m_k, i.e., it is given by $(e_1, ..., e_m, \langle k, t_{-1}, t_{-1} + \delta_k, q\rangle)$ where δ_k^- is the minimum duration of the execution of method m_k. On the other hand, the second representative state is associated with the latest completion time of method m_k, i.e., it is given by $(e_1, ..., e_m, \langle k, t_{-1}, \min\{t_{-1} + \delta_k^+, \Delta_k\}, q\rangle)$ where δ_k^+ is the maximum duration of the execution of method m_k and Δ_k is the latest admissible execution time of method m_k.

In general, the NewRepresentativeState function encounters a situation where the current set $\widetilde{S}_{\phi,s_{-1},q}$ of representative states already contains some representative states (including the two delimiting representative states), i.e., $\widetilde{S}_{\phi,s_{-1},q} = \{(e_1, ..., e_m, \langle k, t_{-1}, t_i, q\rangle)\}_{i=1}^n$. Here, the NewRepresentativeState function adds to $\widetilde{S}_{\phi,s_{-1},q}$ a new representative state $s = (e_1, ..., e_m, \langle k, t_{-1}, t', q\rangle)$ where t' is determined using one of the heuristics below:

Uniform Time Heuristic (H_{UT}) chooses representative states such that they uniformly cover the time range $[t_{-1}+\delta_k^-, \min\{t_{-1}+\delta_k^+, \Delta_k\}]$, i.e., such that the maximum distance between two adjacent representative states is minimized. Formally, for time points $t_0, ..., t_n$ sorted in the increasing order of their values, the H_{UT} chooses t' to be:

$$t' := \frac{t_j + t_{j+1}}{2}$$

Where j is such that the distance $t_{j+1} - t_j$ between time points t_{j+1} and t_j is maximized, that is:

$$j = \arg \max_{i=0,...,n-1} t_{i+1} - t_i.$$

For example, suppose that the set of representative states already contains three representative states with their corresponding times $t_0 = 5$, $t_1 = 10$, and $t_2 = 20$. Here, because $t_2 - t_1 = 10$ which is greater than $t_1 - t_0 = 5$, the H_{UT} heuristic will pick $j = 1$ which will result in adding a new representative state associated with time $t' = (10 + 20)/2 = 15$.

The main advantage of the H_{UT} heuristic is that is it easy to implement. However, the H_{UT} heuristic can in practice be very ineffective because it ignores the likelihood of transitioning to a new representative state s. As a result, the representative states in set $\widetilde{S}_{\phi,s_{-1},q}$ can be situated far away from the states the process transitions to during the execution phase.

Uniform Probability Heuristic (H_{UP}) improves on the H_{UT} heuristic in that it uses the likelihood of transitioning to representative state as a metric for picking a new representative state. The H_{UP} heuristic first estimates[5] the probabilities $F_{\phi,s_{-1},q}(t)$ that the execution of method m_k started in state s_{-1} will be completed *before* time t with outcome q. For notational convenience let $F := F_{\phi,s_{-1},q}(t)$. The H_{UP} heuristic attempts to uniformly cover the probability range $[0, 1]$, i.e., it attempts to minimize the maximum distance between the two adjacent probability values $F(t)$. Formally, for time points $t_0, ..., t_n$ and their corresponding probability values $F(t_0), ..., F(t_n)$ sorted in the increasing order, the H_{UP} chooses t' to be:

$$t' := F^{-1}\left(\frac{F(t_j) + F(t_{j+1})}{2}\right)$$

Where F^{-1} is the inverse function of function F, i.e., $F^{-1}(F(t)) = t$ for all t whereas j is found using the formula below:

$$j = \arg \max_{i=0,...,n-1} F(t_{i+1}) - F(t_i).$$

[5]This estimation uses the Monte-Carlo sampling explained in Section 5.2.4

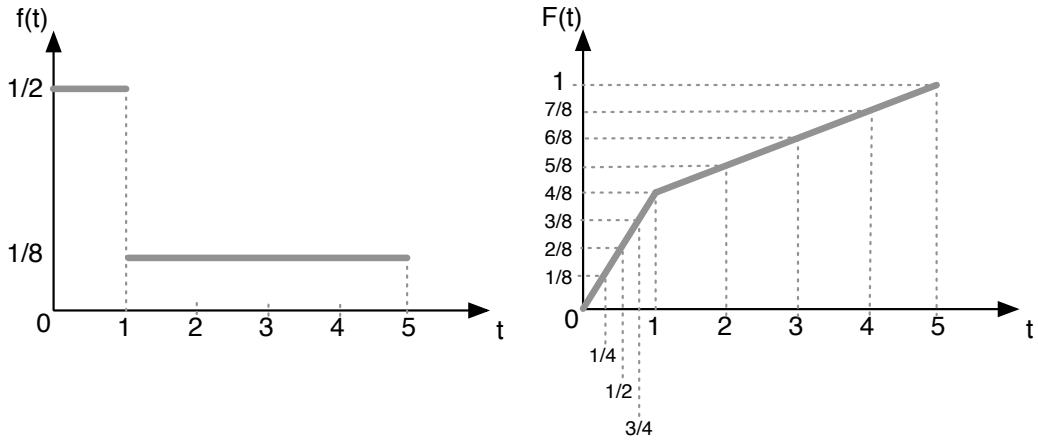

Figure 5.9: Demonstration of the H_{UP} heuristic

To understand how the H_{UP} works in practice and why it is more efficient than the H_{UT} heuristic, consider the situation in Figure 5.9 Here $f := F\frac{d}{dt}$ is a probability density function of the time when the execution of method m_k will terminate. Assume that function f is given by a multi-uniform distribution below: (see Figure 5.9)

$$f(t) = \begin{cases} \frac{1}{2} & \text{if} \quad 0 \leq t < 1 \\ \\ \frac{1}{8} & \text{if} \quad 1 \leq t < 5 \end{cases}$$

Suppose that the H_{UP} heuristic is used to add 9 representative states. The first two representative states are the delimiting representative states that correspond to $t = 0$ and $t = 5$. Now, the heuristic aims to cover the probability range $[0, 1]$ uniformly. First, the representative state at $t = 1$ is added since it splits the probability range $[0, 1]$ into two ranges $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$ of equal length (such splitting minimizes the length of the longest probability range). After adding 6 additional representative states (also in a way that minimizes the length of the longest probability range) all the probability ranges will have the length of $\frac{1}{8}$ and the set of representative states will

contain elements that correspond to times $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 2, 3, 4, 5$ as shown in Figure 5.9. Notice an important phenomena: since it is 4 times more likely that the execution of method m_k will be finished in the time interval $[0, 1)$ than in the time interval $[1, 2)$, the H_{UP} heuristic introduces 4 times more representative states for the time interval $[0, 1)$ than for the time interval $[1, 2)$. As a result, the representative states in set $\widetilde{S}_{\phi,s_{-1},q}$ are more likely to be situated close to the states the process transitions to during the execution phase.

5.2.3.2 Policy for non-Representative States

Prior to developing the formula for the error bound of M-DPFP, it is necessary to explain the *greedy policy* that agents follow when they transition to non-representative states. Intuitively, the greedy policy for an non-representative state s is to execute an action from a representative state $\widetilde{s}$ which is *before* s and *as close as possible to* s. Formally, recall that method sequences ϕ and ϕ_{-1} are such that $\phi = (\phi_{-1}, m_k)$. The greedy policy is then defined as follows: (refer to Figure 5.10)

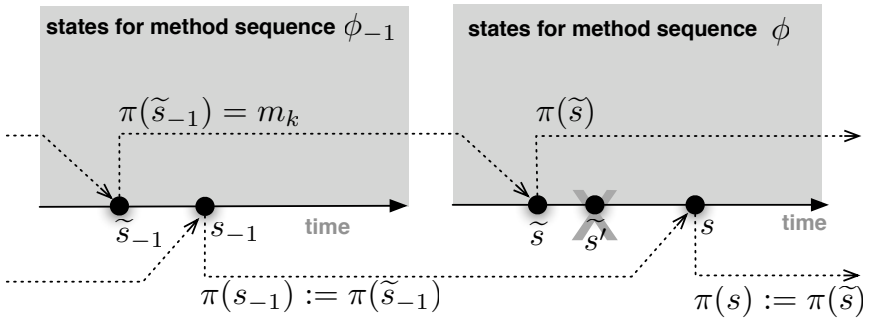

Figure 5.10: Greedy policy for the non-representative states

Definition 7. *The* **greedy policy** *postulates that if an agent:*

- *Was in a non-representative state* $s_{-1} \in S_{\phi_{-1}}$
- *Started in state* s_{-1} *the execution of action* $\pi(\widetilde{s}_{-1}) = m_k$ *where* $\widetilde{s}_{-1} \in \widetilde{S}_{\phi_{-1}}$

- *Transitioned with an outcome q to a non-representative state $s \in S_\phi$*

The agent will start in state s the execution an action $\pi(\widetilde{s})$ where:

- *$\widetilde{s}$ is a representative state from set $\widetilde{S}_{\phi,\widetilde{s}_{-1},q}$*
- *$\widetilde{s}$ is before s, i.e. , if $\widetilde{s} = (..., \langle k, t_{-1}, \widetilde{t}, q\rangle)$ and $s = (..., \langle k, t_{-1}, t, q\rangle)$ then $\widetilde{t} \leq t$*
- *$\widetilde{s}$ is as close as possible to s, i.e., there exists no $\widetilde{s'} \in \widetilde{S}_{\phi,\widetilde{s}_{-1},q}$ such that $\widetilde{s}$ is before $\widetilde{s'}$ and $\widetilde{s'}$ is before s.*

5.2.3.3 Error Bound

In order to bound the error of the M-DPFP algorithm it is necessary to combine the two following errors:

- The error produced by the lookahead function (described in Section 5.2.4)
- The error produced by executing the greedy policy.

Consider the situation in Figure 5.11 where according to the greedy policy, action $\pi^*(\widetilde{s_i}) = m_g$ should be executed in a non-representative state s. The the error of executing in state s a greedy action m_g instead of an optimal action m_o can be bound as follows: Let $\widetilde{s_i}$, $\widetilde{s}_{i+1} \in \widetilde{S}_\phi$ be two adjacent representative states such that state $s \in S_\phi$ is located between $\widetilde{s_i}$ and $\widetilde{s}_{i+1}$. The times at which the agent transitions to states $\widetilde{s_i}$, s, $\widetilde{s}_{i+1}$ are t_i, t_s, t_{i_1} respectively. Futhermore, $t_{i+1} - t_i = \delta$ as shown in Figure 5.11. Also, because states $\widetilde{s_i}$, s, $\widetilde{s}_{i+1}$ correspond to the same execution sequence, the actions applicable in states $\widetilde{s_i}$, s, $\widetilde{s}_{i+1}$ are the same, i.e., $A(s) = A(\widetilde{s_i}) = A(\widetilde{s}_{i+1}) = \{m_j\}_{i\in J}$.

The lookahead function called for a representative state $\widetilde{s_i}$ returns (within the accuracy ϵ_κ) the expected utilities $u_{i,j}$ of executing in state $\widetilde{s_i}$ actions $m_j \in A(\widetilde{s_i})$, (explained in Section 5.2.4).

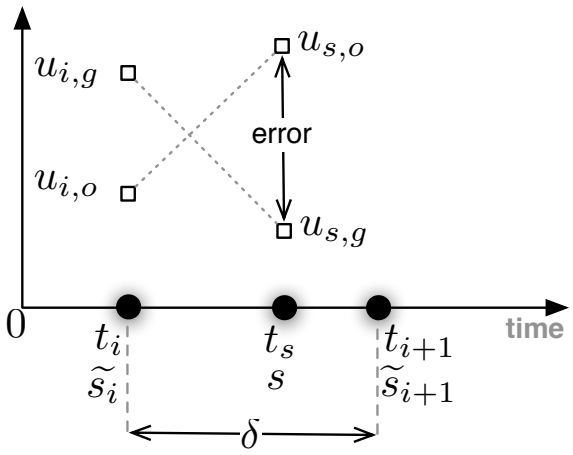

Figure 5.11: Greedy policy and the error bound: Optimal action in state $\widetilde{s_i}$ is to execute method m_g (which yields the utility $u_{i,g}$) whereas the optimal action in state s is to execute method m_o (which yields the utility $u_{s,o}$). Here, the greedy policy for an agent postulates that method m_g should be executed in state s (which yields the utility $u_{s,g}$). The error of the greedy policy for state s is bounded by $|u_{s,o} - u_{s,g}|$.

Observe now, that the expected utilities $u_{s,j}$ of executing actions m_j in state s can be the same, smaller or greater than the corresponding utilities $u_{i,j}$. In particular (refer to Figure 5.11):

- If the likelihood of method m_o being enabled before time t_i is smaller than the likelihood of method m_o being enabled before time $t_s > t_i$ one can have $u_{i,o} < u_{s,o}$.
- Since the likelihood of finishing the execution of method m_g within a time window is greater if the execution of m_g is started in state $\widetilde{s_i}$ rather than in state s one can have $u_{i,g} > u_{s,g}$.

And consequently, the error of the greedy policy for state s is at most $u_{s,o} - u_{s,g}$. To bound this expression, we first find the minimum value of $u_{s,g}$ and then the maximum value of $u_{s,o}$. Notice, that since we are interested in finding the minimum value of $u_{s,g}$ it can be assumed without the loss of generality that method m_g is enabled in state $\widetilde{s_i}$ with probability 1[6]. Recall the probability

[6]If method m_g was enabled in state $\widetilde{s_i}$ with probability $p_1 < 1$ then method m_g would be enabled in state s with probability $p_2 \geq p_1$ and consequently, the difference $u_{i,g} - u_{s_g}$ would be smaller which would translate into smaller error value.

$F_{(\phi,m_g),\widetilde{s_i},q}(t)$ that the execution of method m_g started in state $\widetilde{s_i}$ will finish before time t with result q — the value of $F_{(\phi,m_g),\widetilde{s_i},q}(t)$ can be easily calculated using Monte-Carlo sampling approach explained in Section 5.2.4.2. Now, since the later the execution of method m_g starts, the smaller the likelihood of its completion within an admissible time window. Thus:

$$F_{(\phi,m_g),\widetilde{s_i},q}(t) > F_{(\phi,m_g),s,q}(t) > F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta)$$

And since the expected utility $u_{s,g}$ is directly affected by $F_{(\phi,m_g),s,q}(t)$ it holds that:

$$u_{s,g} > u_{i,g} \cdot (1 - (F_{(\phi,m_g),\widetilde{s_i},q}(t) - F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta)))$$

Conversely, it is now shown how to establish the maximum value of $u_{s,o}$. Notice, that since we are interested in finding the maximum value of u_{s_o} it can be assumed without the loss of generality that method m_o will be completed within its admissible time window with probability 1, no matter if it is started in state $\widetilde{s_i}$ or s[7]. Then, using Monte-Carlo sampling approach explained in Section 5.2.4.2 one can calculate the probabilities $F^e_{i,o}(t)$ that method m_o will be enabled before time t when the agent is in state $\widetilde{s_i}$ ($F^e_{i,j}$ is a cumulative distribution function). Finally, since the later the execution of method m_o starts, the higher the probability that it will be enabled. Thus:

$$F^e_{i,o}(t) < F^e_{s,o}(t) < F^e_{i,o}(t+\delta)$$

[7]If method m_o is completed within its admissible time window with probability $p_1 < 1$ when it is started in state $\widetilde{s_i}$ then this probability can only get smaller if the execution of m_o is started in state s and consequently, the value of u_{s_o} and the error can only get smaller.

And since the expected utility $u_{s,o}$ is directly affected by $F^e_{s,o}(t)$ it holds that:

$$u_{s,o} < u_{i,o} \cdot (1 + (F^e_{i,o}(t+\delta) - F^e_{i,o}(t)))$$

Consequently, the error of executing a greedy action m_g instead of an optimal action m_o in state s is bounded by:

$$\begin{aligned} u_{s,o} - u_{s,g} &< u_{i,o}(1 + (F^e_{i,o}(t+\delta) - F^e_{i,o}(t))) - u_{i,g}(1 - (F_{(\phi,m_g),\widetilde{s_i},q}(t) - F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta))) \\ &< u_{i,g}(F^e_{i,o}(t+\delta) - F^e_{i,o}(t) + F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta) - F_{(\phi,m_g),\widetilde{s_i},q}(t)) \end{aligned}$$

Incorporating the error ϵ_κ of the lookahead function in determining $u_{s,o}$ and $u_{s,g}$:

$$< u_{i,g}(F^e_{i,o}(t+\delta) - F^e_{i,o}(t) + F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta) - F_{(\phi,m_g),\widetilde{s_i},q}(t)) + 2\epsilon_\kappa \tag{5.11}$$

And since m_o is not known, the error ϵ_i produced by following a greedy policy from a state between the representative states $\widetilde{s_i}$ and $\widetilde{s_{i+1}}$ that are δ apart is given by[8]:

$$\epsilon_i < \max_{m_o \in A(\widetilde{s_i})} u_{i,g}(F^e_{i,o}(t+\delta) - F^e_{i,o}(t) + F_{(\phi,m_g),\widetilde{s_i},q}(t-\delta) - F_{(\phi,m_g),\widetilde{s_i},q}(t)) + 2\epsilon_\kappa \tag{5.12}$$

Finally, for a given set $\widetilde{S}_\phi$ of representative states for the method sequence $\phi = (\phi_{-1}, m_k)$, the error of the M-DPFP algorithm for that sequence, denoted as computeError($\widetilde{S}_\phi$), is bounded by:

$$\text{computeError}(\widetilde{S}_\phi) < \max_{\substack{s_{-1} \in \widetilde{S}_{\phi_{-1}} \\ \pi(s_{-1}) = m_k \\ q \in \{0,1\}}} \max_{\widetilde{s_i} \in \widetilde{S}_{\phi,s_{-1},q}} \epsilon_i.$$

[8] Assuming that representative states $\widetilde{s_i}$ and $\widetilde{s_{i+1}}$ exist

To summarize, the M-DPFP algorithm (see Algorithm 5.2.2) is called with a desirably small parameter *maxError*. The algorithm then keeps adding the representative states until COMPUTEERROR ($\widetilde{S}_\phi$) is smaller than *maxError* for any sequence ϕ that can be encountered while executing the best policy. Finally, since the total number of sequences that can be encountered by the agent team while executing a policy is bounded by the number K of methods, the cumulative error of the M-DPFP algorithm is bounded by $K \cdot maxError$.

This section demonstrated how to select new representative states, what actions agents should execute in non-representative states and how to bound the error of the M-DPFP algorithm. The next section shows how the lookahead function determines which action should be executed from a representative state.

5.2.4 Lookahead Function

This section describes the lookahead function OPTIMALMETHOD(s) called by Algorithm 5.2.2 in Section 5.2.2. The lookahead function takes as an input the representative state s and returns the expected utilities $u_{s,i}$ of executing actions $m_i \in A(s)$ from state s. In order to calculate $u_{s,i}$ the lookahead function solves a dual problem, i.e., it tries to identify an optimal probability mass flow through states (of all agents) starting from state s that maximizes the total expected utility $u_{s,i}$.

As stated, there are clear similarities between the lookahead function of M-DPFP and the DPFP algorithm from Section 3.2, however there is one key difference: Whereas DPFP could perform the forward search in the space associated with the future states (methods) of a single agent, M-DPFP's search must consider future methods of possibly all agents. In other words, M-DPFP's lookahead function called for state s of agent n must consider future methods of agent n

and future methods of agents $n' \neq n$ that are affected by the execution of future methods of agent n. Hence, M-DPFP's lookahead function must know the probabilities that agents $n' \neq n$ will start the execution of their future methods and in order to estimate these probabilities, policies of agents $n' \neq n$ must be found first. M-DPFP achieves that by adding the necessary representative states (and finds policies for these representative states) for agents $n' \neq n$ prior to considering state s of agent n. This process in explained in detail in the next three sections.

5.2.4.1 Dual Problem Formulation

Before the dual problem formulation in context of the lookahead function is used, it is necessary to develop the dual formulation in the general case. Let $\phi = (m_{i_1}..., m_{i_k})$ denote a sequence of k methods that an agent has executed and $Q = (q_{i_1}, ..., q_{i_k})$ be the vector of k outcomes of method execution where q_i is the outcome of the execution of method m_i. For notational convenience, let $A(\phi)$ be the set of methods that an agent can execute after it completed (successfully or unsuccessfully) the execution of methods from sequence ϕ. Furthermore, let $F^{\pi}_{\phi,Q}(t)$ be the probability that the agent has completed before time t the execution of methods from sequence ϕ with outcomes Q when following a policy π and $F^{\pi}_{\phi,Q}(m_l)(t)$ be the probability that the agent has completed the execution of methods from sequence ϕ with outcomes Q and started the execution of method m_l

before time t when following a policy π — observe that $F^{\pi}_{\phi,Q}$ and $F^{\pi}_{\phi,Q}(m_l)$ are cumulative distribution functions over $t \in [0, \Delta]$. The expected joint reward that the agents receive for following the policy π from their starting states $s_{0,n}$ $n = 1, ..., N$ is then given by:

$$\begin{aligned} V^{\pi}(s_{0,1}, ..., s_{0,n}) &= \sum_{n=1,...,N} \sum_{\substack{\phi=(m_{i_1},...,m_{i_k})\in\Phi_n \\ Q=(q_{i_1},...,q_{i_k})\in\{0,1\}^k}} F^{\pi}_{\phi,Q}(\Delta) \cdot q_{i_k} \cdot r_{i_k} \\ &= \sum_{\substack{\phi=(m_{i_1},...,m_{i_k})\in\cup_n\Phi_n \\ Q=(q_{i_1},...,q_{i_k})\in\{0,1\}^k}} F^{\pi}_{\phi,Q}(\Delta) \cdot q_{i_k} \cdot r_{i_k} \end{aligned} \tag{5.13}$$

In particular, for an optimal policy π^* the joint expected reward $V^{\pi^*}(s_{0,1}, ..., s_{0,n})$ (denoted for notational convenience as $V^*(s_{0,1}, ..., s_{0,n})$) is maximized:

$$V^*(s_{0,1}, ..., s_{0,n}) = \max_{\pi} V^{\pi}(s_{0,1}, ..., s_{0,n}) = \sum_{\substack{\phi=(m_{i_1},...,m_{i_k})\in\cup_n\Phi_n \\ Q=(q_{i_1},...,q_{i_k})\in\{0,1\}^k}} F^{*}_{\phi,Q}(\Delta) \cdot q_{i_k} \cdot r_{i_k} \tag{5.14}$$

Let $F^{\pi} := \{F^{\pi}_{\phi,Q}$ such that $\phi = (m_{i_1}, ..., m_{i_k}) \in \Phi$ and $Q \in \{0,1\}^k\}$ be a set of cumulative distribution functions associated with a policy π and $F^* := \{F^{*}_{\phi,Q}$ such that $\phi = (m_{i_1}, ..., m_{i_k}) \in \Phi$ and $Q \in \{0,1\}^k\}$ be a set of optimal cumulative distribution functions associated with the optimal policy π^*. In order to find F^* observe that the cumulative distribution functions in F^* must

maximize the right-hand-side of Equation (5.13). Furthermore, F^* must be an element of a set $X = \{F : (5.15), (5.16), (5.17)\}$ where constraints (5.15), (5.16), (5.17) are defined as follows:[9]

$$F_{(m_{-n}),(1)}(t) = 1 \tag{5.15}$$

$$F_{\phi,Q}(t) = \sum_{m_l \in A(\phi)} F_{\phi,Q}(m_l)(t) + F_{\phi,Q}(m_0)(t) \tag{5.16}$$

$$F_{(\phi,m_l),(Q,q)}(t) = \int_0^t Pb(m_l, t', q) \cdot F_{\phi,Q}(m_l)(t') p_l(t - t') dt' \tag{5.17}$$

for all agents $n = 1, ..., N$, sequences $\phi \in \Phi_n$ and outcomes $Q \in \{0, 1\}^{|\phi|}$. Here, (ϕ, m_l) is a concatenation of ϕ and (m_l), (Q, q) is a concatenation of Q and (q) whereas $Pb(m_l, t', q)$ (explained later) is the probability that method m_l is enabled before time t' (in case $q = 1$) or *not* enabled before time t' (in case $q = 0$). Constraints (5.15), (5.16), (5.17) are explained as follows:

- Constraint (5.15) ensures that each agent $n = 1, .., N$ starts the execution from its starting state $s_{0,n}$. To observe that, recall that the starting state $s_{n,0}$ of agent n in encoded in the CR-DEC-MDP framework as $s_{n,0} = (\langle -n, l_{n,0}, l_{n,0}, 1 \rangle)$, that is, a spoof method m_{-n} is by default completed successfully (with $q = 1$) at time $l_{n,0} = 0$. Hence the probability $F_{(m_{-n}),(1)}(t)$ that method m_{-n} will be completed successfully before time t must be 1 for all $t \in [0, \Delta]$.

- Constraint (5.16) can be interpreted as the conservation of of probability mass flow through a method sequence ϕ. Applicable only if $|A(\phi)| > 0$ it ensures that the cumulative distribution function $F_{\phi,Q}$ is split into cumulative distribution functions $F_{\phi,Q}(m_l)$ for methods

[9] Constraints (5.15), (5.16), (5.17) are defined for a single method execution time windows $[0, \Delta]$. Extension to multiple time windows is shown in Equation (5.6)

$m_l \in A(\phi)$ plus the cumulative distribution function $F_{\phi,Q}(m_0)$ — this latter function represents the probability that the agent will wait (do nothing) upon completing the execution of methods from sequence ϕ with outcomes Q.

- Constraint (5.17) ensures the correct propagation of probability mass from a method sequence ϕ to a method sequence (ϕ, m_l) upon executing method m_l with an outcome q. For $q = 1$, the constraint ensures that the execution of method m_l will be finished *successfully* at time t if this execution: (i) is started at time t', (ii) takes time $t - t'$ to complete and (iii) was started after method m_l had been enabled. In contrast, for $q = 0$, constraint (5.17) ensures that the execution of method m_l will be finished *unsuccessfully* at time t if this execution: (i) is started at time t', (ii) takes time $t - t'$ to complete and (iii) was started when method m_l was not enabled. The correct implementation of constraint (5.17) requires that the probabilities $Pb(m_l, t', q)$ are known — in Section 5.2.4.2 it is explained how to estimate these probabilities using a Monte-Carlo sampling approach.

The dual problem is then stated as:

$$\begin{aligned} \max \quad & \sum_{\substack{\phi=(m_{i_1},\ldots,m_{i_k})\in\cup_n\Phi_n \\ Q=(q_{i_1},\ldots,q_{i_k})\in\{0,1\}^k}} F_{\phi,Q}(\Delta)\cdot q_{i_k}\cdot r_{i_k} \\ s.t. \quad & F \in X \end{aligned} \tag{5.18}$$

And the solution to the dual problem is a set F^* which yields the total expected utility $V^*(s_{0,1}, ..., s_{0,n})$.

5.2.4.2 Dual Problem and the Lookahead Function

It is now shown how the lookahead function takes advantage of the dual problem formulation. Recall, that the lookahead function takes as an input the representative state s and returns the expected utilities $u_{s,k}$ of executing actions $m_k \in A(s)$ from state s. It is first shown how the lookahead function uses the dual problem formulation in a simple case, when s is one of the starting states of the CR-DEC-MDP model. This result is then generalized to an arbitrary representative state s.

Consider a situation when $s = s_{0,n}$, i.e., s is a starting state and $A(s_{0,n}) = \{m_1, ..., m_K\}$ as shown in Figure 5.12a. In order to determine which method should agent n start executing from state $s_{0,n}$ one must consider K separate cases. In particular, for case k, the dual problem (5.18) is solved with an additional constraint $F_{(m_{-n})(1)}(m_k)(t) = 1$ for $t \in [0, \Delta]$ which ensures that method m_k is going to be executed from state s. The solution to the new problem then determines the expected utility $u_{s,k}$ of executing method m_k in state s. Upon considering all cases $k = 1, ..., K$ the lookahead function returns method m_{k^*} where $k^* = \arg\max_{k=1,...,K} u_{s,k}$.

Consider now a more general case (refer to Figure 5.12b) when the lookahead function is called to find the method to be executed from a representative state $s \in S_\phi$ of agent n. Assume that methods from the sequence ϕ have been executed with outcomes Q and $A(s) = A(\phi) = \{m_1, ..., m_K\}$. Furthermore, let $D(\phi)$ be the set of method sequences $\phi' \in \cup_{n=1,...,N}\Phi_n$ (of any agent) that are *preceded* by the method execution sequence ϕ, that is, that are reachable from node ϕ of the method sequences graph G (see Section 5.2.2). For example, in Figure 5.12b we have $D((m_{-2}, m_4, m_6)) = \{(m_{-2}, m_4, m_6, m_5), (m_{-1}, m_1, m_2), (m_{-1}, m_1, m_2, m_3), (m_{-1}, m_1, m_3, m_2), (m_{-1}, m_3, m_2, m_1)\}$.

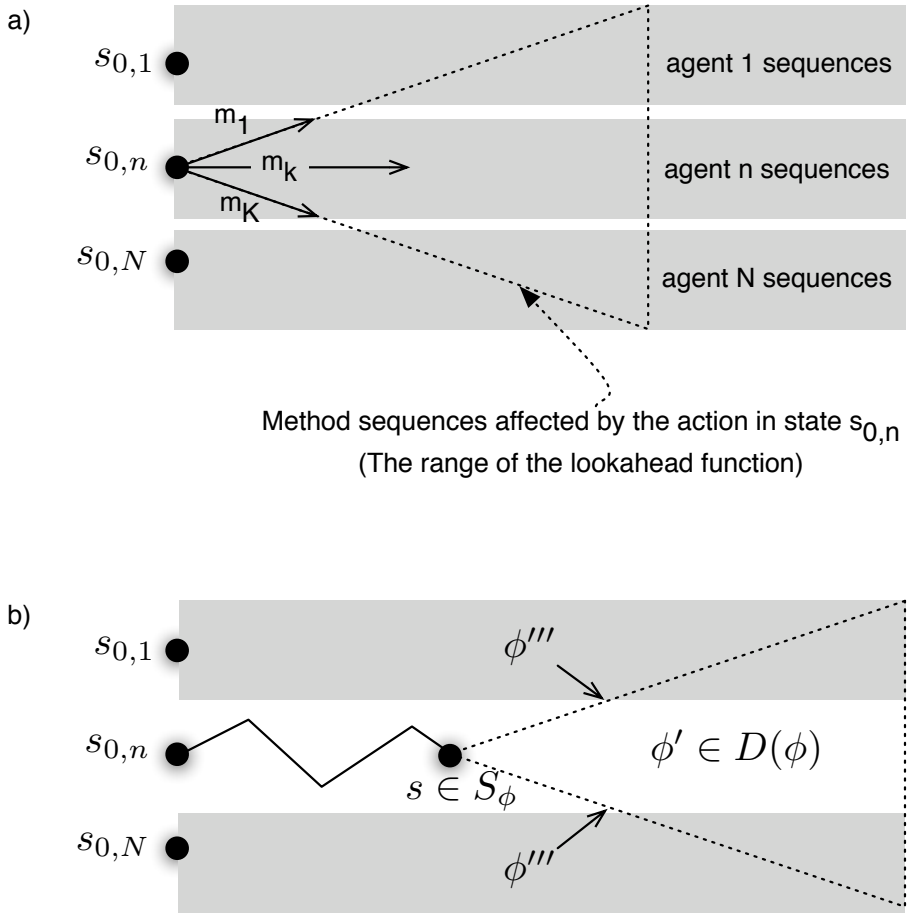

Figure 5.12: The range of the lookahead function: (a) Gray rectangles encompass all method execution sequences of one agent; the dotted triangle encompasses the range of the lookahead function, i.e., all the sequences that are affected by the decision of agent n in the initial state $s_{0,n}$. (b) When the lookahead function is called for state $s \in S_\phi$ of agent n, $D(\phi)$ is the set of sequences ϕ' affected by the agent decision in state s. The execution of methods in sequences ϕ''' affect the execution of methods in sequences $\phi' \in D(\phi)$.

Observe that the action that agent n executes in state s (for sequence ϕ) will have an affect on the execution outcome of other methods (that possibly belong to different agents). Precisely, action in state s affects all the methods $m \in \cup_{n=1,..,N} M_n$ such that there is a sequence $\phi' = (..., m) \in$

$D(\phi)$. In other words, the action that the agent executes in state s will impact all the functions $F_{\phi',Q'}$ for $\phi' \in D(\phi)$. Thus, the objective function of the agent in state s is the following:

$$\max \sum_{\substack{\phi'=(m_{i_1},\ldots,m_{i_k})\in D(\phi) \\ Q'=(q_{i_1},\ldots,q_{i_k})\in\{0,1\}^k}} F_{\phi',Q'}(\Delta) \cdot r_{i_k} \cdot q_{i_k}. \tag{5.19}$$

Furthermore, agent n must consider the impact of methods from external external sequences (from outside of $D(\phi)$) when maximizing the value of the expression 5.19. Precisely, the fact that the agent has reached state s imposes additional constraints on the dual problem formulation (5.18), i.e., set X specified by constraints (5.15), (5.16), (5.17) is now further constrained by:

- *Local agent history constraints*: Because the state s is fully observable to the agent, the agent knows exactly the starting times, finishing times and outcomes of the execution of all its methods of sequence ϕ. For example, if agent 1 is in state $s = (\langle -1, t_0, t_0, 1\rangle, \langle 1, t_0, t_1, 1\rangle, \langle 2, t_1, t_2, 0\rangle)$ then it can infer that $F_{(m_{-1}),(1)}(t) = 1$ for $t \in [0, \Delta]$; $F_{(m_{-1},m_1),(1,1)}(t) = 0$ if $t \in [0, t_1]$ and 1 if $t \in [t_1, \Delta]$; $F_{(m_{-1},m_1,m_2),(1,1,1)}(t) = 0$ for $t \in [0, \Delta]$ and finally $F_{(m_{-1},m_1,m_2),(1,1,0)}(t) = 0$ if $t \in [0, t_2]$ and 1 if $t \in [t_2, \Delta]$. In general, when methods from a sequence ϕ are completed, functions $F_{\phi'',Q''}$ where ϕ'' is a prefix of ϕ can be inferred. These functions $F_{\phi'',Q''}$ specify new constraints to be added to further restrict set X.

- *Other agents histories constraints*: Because the information encoded in state s does not allow the agent to determine precisely the current states of other agents, the agent can only estimate the starting times, finishing times and outcomes of the execution of methods of other agents. Of particular importance for solving the dual problem of the agent (when the agent is about to choose an action to be executed in state s) are the method execution

sequences ϕ''' of other agents that can affect the execution of methods from sequences $\phi' \in D(\phi)$ and thus, influence the decision of the agent in state s. Precisely, in state s the agent must be able to estimate the status of execution of methods from sequences ϕ''' that *precede* method sequences $\phi' \in D(\phi)$, i.e., sequences ϕ''' of other agents such that there exists a path $(\phi''', ..., \phi')$ for some $\phi' \in D(\phi)$ in the method sequences graph G.

In order to determine $F_{\phi''',Q'''}(t)$ for $t \in [0, \Delta]$ for all such sequences, a Monte-Carlo sampling approach is used [MacKay, 1998]. In essence, the CR-DEC-MDP model assumes given starting states $s_{0,1}, ..., s_{0,N}$ of the agents. Moreover, thanks to the specific ordering $\mathcal{L}$ in which the lookahead function analyses the method execution sequences, when the M-DPFP algorithm calls the lookahead function for state $s \in \widetilde{S}_{\phi}$, the policies for states $s''' \in S_{\phi'''}$ are known (if s''' is not a representative state, the greedy policies are used — refer to Section 5.2.3.2). Thus, a Monte-Carlo algorithm initiated with a given number of samples (varied in Chapter 6) can simulate the concurrent execution of agents policies from their starting states. By counting the number of samples that traverse a particular sequence ϕ''' with outcomes Q''' in a given time interval, one can estimate the probabilities $F_{\phi''',Q'''}(t)$ for all $t \in [0, \Delta]$. The functions $F_{\phi''',Q'''}$ are then used to specify new constraints to be added to further restrict set X.

To summarize, in order to determine which method should agent n start executing from state $s = (..., \langle m_l, t_1, t_2, q \rangle) \in \widetilde{S}_{\phi}$ the lookahead function considers K separate cases. In particular, for case k the lookahead function instantiates a new dual problem with the objective function (5.19) and constraints specified by (5.15), (5.16), (5.17), local agent history constraints, other agent histories constraints and an additional constraint $F_{\phi,Q}(m_k)(t) = 0$ for $t \in [0, t_2]$ and $F_{\phi,Q}(m_k)(t) =$

1 for $t \in [t_2, \Delta]$ which ensures that method m_k is going to be executed from state s . The solution to the dual problem then determines the expected utility $u_{s,k}$ of executing method m_k in state s. Upon considering all cases $k = 1, ..., K$ the lookahead function returns method m_{k^*} where $k^* = \arg\max_{k=1,...,K} u_{s,k}$. Finally, if the maximum utility of executing an action in a representative state $s = (..., \langle m_l, t_1, t_2, q\rangle)$ is smaller than the maximum utility of executing an action in a *later* representative state, i.e. $s' = (..., \langle m_l, t_1, t_2 + t', q\rangle)$ then the optimal policy in the representative state s is to wait.

5.2.4.3 Solving the Dual Problem

In order to solve the dual problem above, The same approach as in the DPFP algorithm in Section 3.2.3 can be used. Informally, each cumulative distribution function $F_{\phi,Q}$ is approximated with a step function of a step height κ. With this approximation technique the feasible set X of the dual problem is narrowed to a restricted feasible set $\widehat{X} \subset X$ which contains a finite number of elements. The search for the optimal solution $\widehat{F}^*$ to the *restricted dual problem* can then be implemented as an exhaustive iteration over all the elements of $\widehat{X}$, to find an element that maximizes the objective function (5.19).

At a basic level, the exhaustive iteration starts with a step function $F_{\phi,Q}$ which is known to the agent (when the agent is in state s) and then considers all possible ways in which $F_{\phi,Q}$ can be split into functions $F_{\phi,Q}(m_i)$ for $m_i \in A(\phi) \cup \{m_0\}$ — spoof method m_0 is used to implement the waiting action of the agent. The algorithm then enters a recursive loop; it fixes functions $\{F_{\phi,Q}(m_i)\}_{m_i \in A(\phi)}$ and considers a method sequence ϕ' such that $\phi' \in D(\phi)$ and ϕ' is the next element on the list $\mathcal{L}$ after ϕ. For sequence ϕ', the algorithm considers all possible ways in which function $F_{\phi',Q'}$ (derived using Monte-Carlo sampling technique discussed above) can be split into

functions $F_{\phi',Q'}(m_j)$ for $m_j \in A(\phi') \cup \{m_0\}$. Here, the algorithm enters another recursive loop (2-*nd* level of recursion); it fixes functions $\{F_{\phi',Q'}(m_j)\}_{m_j \in A(\phi')}$ and considers a method sequence $\phi'' \in D(\phi')$ such that ϕ'' is the next element on the list $\mathcal{L}$ after ϕ' etc. The recursion stops after all possible elements of $\widehat{X}$ have been considered and evaluated using the objective function (5.19). At this point, the optimal solution $\widehat{F}^*$ to the restricted dual problem is known.

It is guaranteed that the the reward error ϵ_κ of a policy identified by $\widehat{F}^*$ can be expressed in terms of κ. Indeed, the error ϵ_κ of the lookahead function can be bounded in exactly the same way as the error of the DPFP algorithm: (refer to Section 3.2.5)

$$\begin{aligned} \epsilon_\kappa &= R_{max} \sum_{\phi' \in D(\phi)} \max_{t \in [0,\Delta]} |F^*_{\phi',Q}(t) - \widehat{F}^*_{\phi',Q}(t)| \\ &\leq \kappa R_{max} \sum_{\phi' \in D(\phi)} |\phi'| \end{aligned}$$

Where $R_{max} = \max_{m_i \in M} r_i$. Hence, by decreasing κ, the lookahead function can trade off speed for optimality.

Chapter 6: Experiments with Multiagent Algorithms

This chapter reports on the empirical evaluation of the algorithms for solving CR-DEC-MDPs. The Chapter is composed of two sections: First, in Section 6.1 the evaluation of the Value Function Propagation (VFP) algorithm introduced in Section 5.1 is presented. VFP is compared with its closest competitor, the Opportunity Cost, Decentralized MDP (OC-DEC-MDP) algorithm [Beynier and Mouaddib, 2006]. Then, in Section 6.2 the evaluation of the Multiagent Dynamic Probability Function Propagation (M-DPFP) algorithm is conducted. M-DPFP is shown to successfully trade-off optimality for speed when solving problems modeled as CR-DEC-MDPs.

6.1 VFP Experiments

First, the experimental evaluation of the VFP algorithm is provided. The VFP algorithm has been developed to provide two orthogonal improvements over the OC-DEC-MDP algorithm [Beynier and Mouaddib, 2006]: (i) VFP uses functional representation to speed up the search for policies carried out by OC-DEC-MDP and (ii) VFP implements a family of opportunity cost splitting heuristics, to allow to find solutions of higher quality. Both algorithms were run to find locally optimal solutions to a special case of CR-DEC-MDPs which assume that methods are fully ordered (see Section 5.1). Furthermore, method execution durations were assumed to follow discrete

probability density function because the OC-DEC-MDP algorithm does not allow for continuous probability density functions of method execution durations.

Hence, the experimental evaluation of VFP consists of two parts: Section 6.1.1 provides a comparison of the solution quality found by either OC-DEC-MDP or VFP when running with different opportunity cost splitting heuristics. Then, Section 6.1.2 reports on the evaluation of the efficiency of both algorithm for a variety of CR-DEC-MDP configurations.

6.1.1 Evaluation of the Opportunity Cost Splitting Heuristics

The VFP algorithm was first run on a generic mission plan configuration from Figure 5.5 in Section 5.1.5 where only methods m_{j_0}, m_{i_1}, m_{i_2} and m_0 were present. Time windows of all methods were set to 400, duration p_{j_0} of method m_{j_0} was uniform, i.e., $p_{j_0}(t) = \frac{1}{400}$ and durations p_{i_1}, p_{i_2} of methods m_{i_1}, m_{i_2} were normal distributions, i.e., $p_{i_1} = N(\mu = 250, \sigma = 20)$, and $p_{i_2} = N(\mu = 200, \sigma = 100)$. Furthermore, only method m_{j_0} provided reward, i.e. $r_{j_0} = 10$ was the reward for finishing the execution of method m_{j_0} before time $t = 400$. The results are shown in Figure (6.1.1) where the x-axis of each of the graphs represents time whereas the y-axis represents the opportunity cost. The first graph confirms that when the opportunity cost function O_{j_0} was split into opportunity cost functions O_{i_1} and O_{i_2} using the $H_{\langle 1,1\rangle}$ heuristic, the function $O_{i_1} + O_{i_2}$ was not always below the O_{j_0} function (the opportunity cost O_{j_0} function was overestimated). In particular, $O_{i_1}(280) + O_{i_2}(280)$ exceeded $O_{j_0}(280)$ by 69%. When heuristics $H_{\langle 1,0\rangle}$, $H_{\langle 1/2,1/2\rangle}$ and $\widehat{H}_{\langle 1,1\rangle}$ were used (graphs 2,3 and 4), the function $O_{i_1} + O_{i_2}$ was always below O_{j_0}.

Then, the experiments in the the civilian rescue domain introduced in Chapter 2 were conducted. For a CR-DEC-MDP configuration from Figure 2.3 all method execution durations were sampled from the normal distribution $N = (\mu = 5, \sigma = 2)$). To obtain the baseline for the heuristic

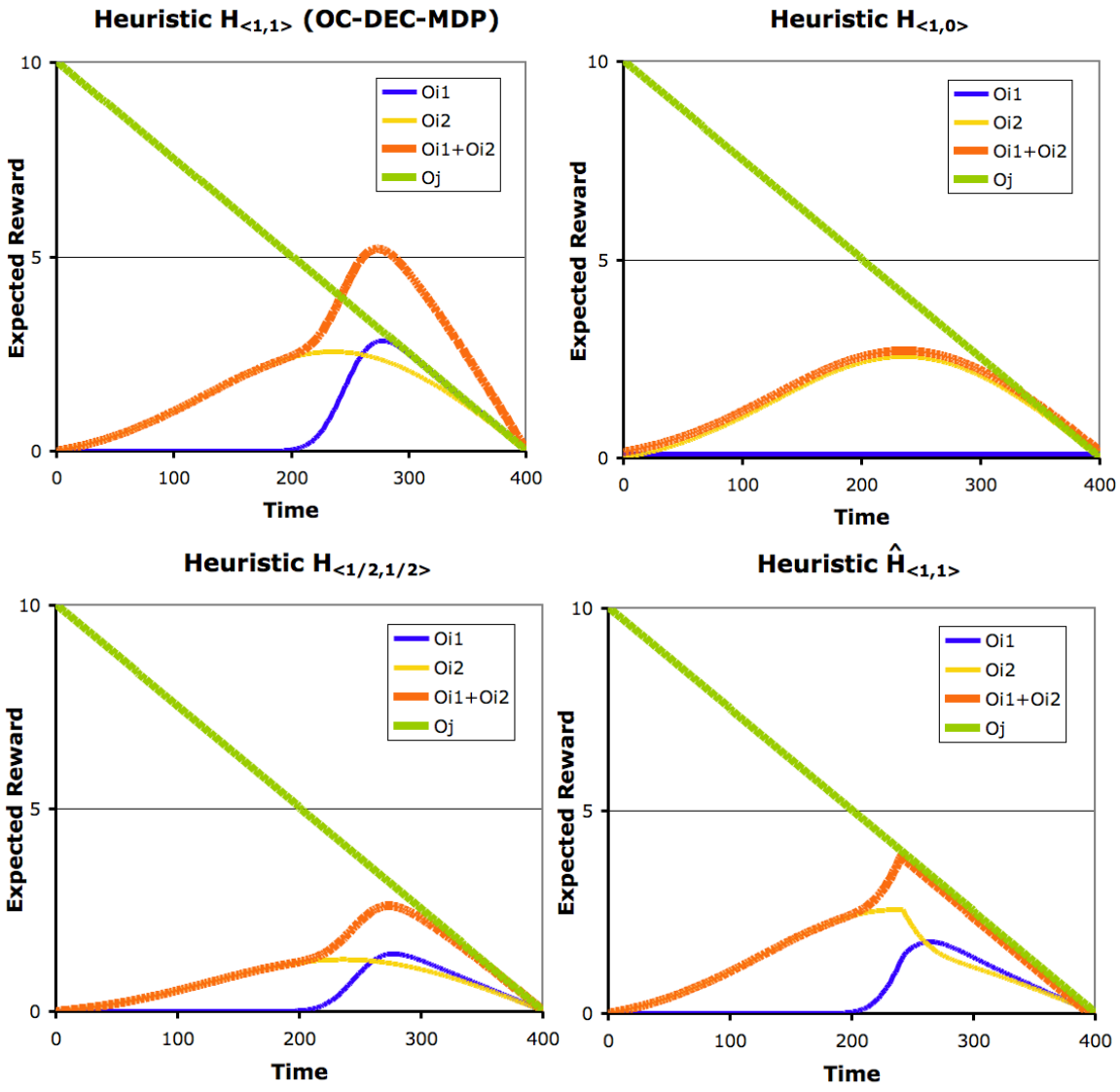

Figure 6.1: Visualization of the opportunity costs splitting heuristics

performance, a globally optimal solver has been implemented — it found a true expected total reward for this domain (Figure (6.3a)). This reward was then compared with a expected total reward found by a locally optimal solver guided by each of the discussed heuristics. Figure (6.3a), which plots on the y-axis the expected total reward of a policy complements the previous results: $H_{\langle 1,1 \rangle}$ heuristic overestimated the expected total reward by 280% whereas the other heuristics were able to guide the locally optimal solver close to a true expected total reward.

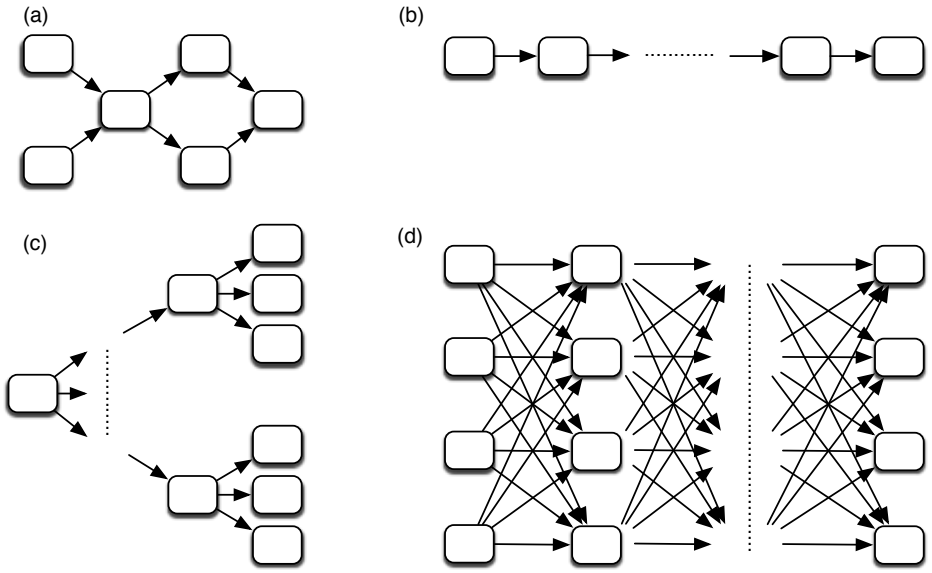

Figure 6.2: Mission plan configurations

6.1.2 Comparison of VFP and OC-DEC-MDP Efficiency

The efficiency VFP and OC-DEC-MDP was then compared when the algorithms were run on configurations shown in Figure 6.2. Both algorithms were using the $H_{\langle 1,1 \rangle}$ heuristic to split the

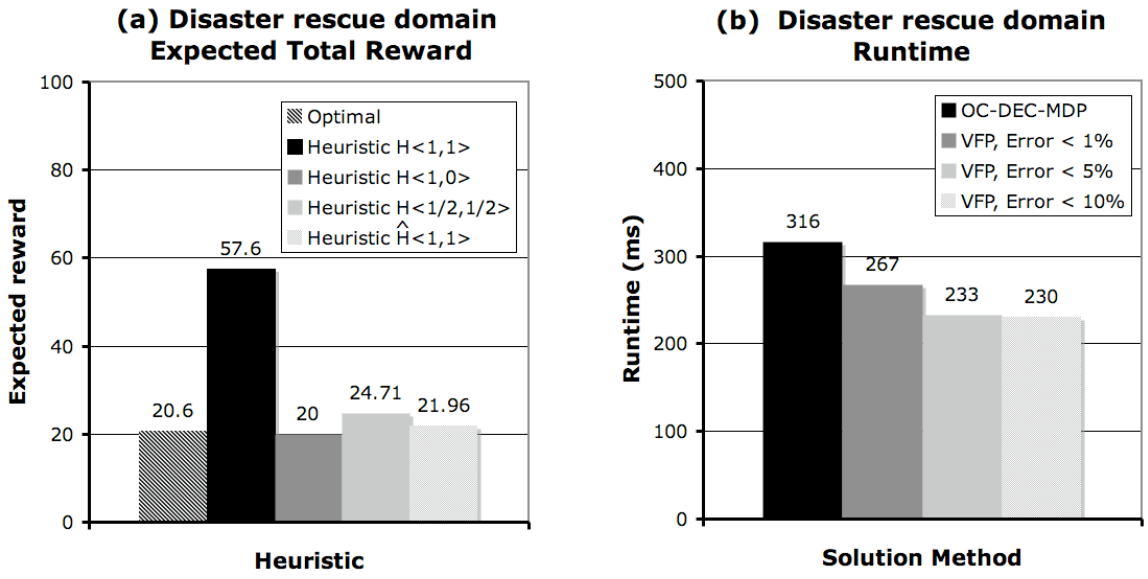

Figure 6.3: VFP performance in the civilian rescue domain.

opportunity cost functions. Using the performance of the OC-DEC-MDP algorithm as a benchmark the VFP scalability experiments were started on a configuration from Figure (6.2a) for which the method execution durations were extended to normal distributions $N(\mu = 30, \sigma = 5)$ and the mission deadline was extended to $\Delta = 200$.

The efficiency of VFP when it was running with three different levels of accuracy was then tested. Different approximation parameters ϵ_P and ϵ_V were chosen, such that the total error of the solution found by VFP stayed within 1%, 5% and 10% of the solution found by OC- DEC-MDP. Both algorithms were run for a total of 100 policy improvement iterations.

Figure (6.3b) shows the performance of VFP in the civilian rescue domain (y-axis shows the runtime in milliseconds). As can be seen, for this small domain, VFP runs 15% faster than OC-DEC-MDP and finds a policy less with an error of less than 1%. For comparison, the globally optimal solved did not terminate within the first three hours of its runtime which shows the potential of the approximate algorithm like OC-DEC-MDP and VFP.

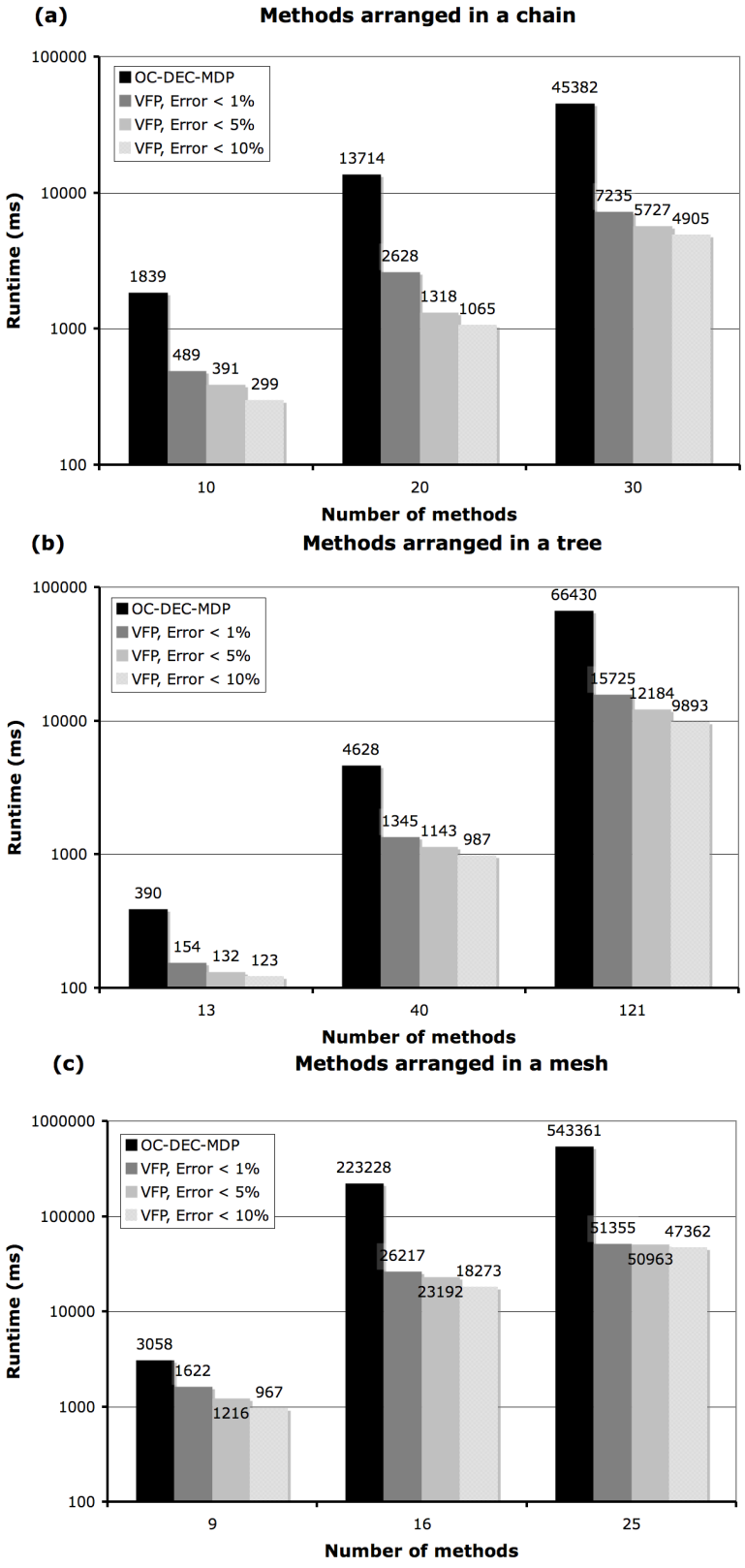

Figure 6.4: Scalability experiments for OC-DEC-MDP and VFP for different CR-DEC-MDP configurations.

Next, VPF was evaluated in a more challenging domain, i.e., with methods forming a long chain (Figure (6.2b)). The experiments with chains of 10, 20 and 30 methods were conducted, with the method execution time windows extended to 350, 700 and 1050 time ticks[1] , to ensure that later methods are reachable. The results are shown in Figure (6.4a), where the number of methods is varied on the x-axis and the algorithm runtime is marked on the y-axis (notice the *logarithmic* scale). As can be observed, the increase of the size of the domain reveals high performance of VFP: Within 1% error, it runs up to 6 times faster than OC-DEC-MDP.

Then, the VFP scalability experiments with methods arranged in a tree have been conducted (Figure (6.2c)). In particular, trees with branching factors of 3 have been considered, with the tree depth of 2, 3 and 4. To ensure that methods have a change to be executed, the time horizon was increased from 200 to 300, and then to 400 time ticks. The results are shown in Figure (6.4b). Although the speedups are smaller than in case of a chain, VFP still runs up to 4 times faster than OC-DEC-MDP when computing the policy with an error of less than 1%.

Finally, VFP was tested on a domain with methods arranged in a $n \times n$ mesh, i.e., $C_{\prec} = \{\langle m_{i,j}, m_{k,j+1}\rangle\}$ for $i = 1, ..., n;\ k = 1, ..., n;\ j = 1, ..., n-1$. In particular, meshes of 3×3, 4×4, and 5×5 methods have been considered. For such configurations the time horizons were increased from 3000 to 4000, and then to 5000, to ensure that all methods have a chance to be executed. The final set of results is shown in Figure (6.4c). As can be seen, especially for larger meshes, VFP runs up to one order of magnitude faster than OC-DEC-MDP while finding a policy that is within less than 1% from the policy found by OC- DEC-MDP.

[1]Recall, that OC-DEC-MDP requires time to be discretized

6.2 M-DPFP Experiments

This section reports on the empirical evaluation of the M-DPFP algorithm, developed to provide error bounded solutions to problems modeled as CR-DEC-MDPs. M-DPFP has been implemented and tested on the domain described in Section 2.1.3.2. All method execution durations were sampled from the Normal distribution $\mathcal{N}(\mu = 2; \sigma = 1)$, method execution time windows and method rewards were $[0, 10]$, $[0, 7]$, $[0, 8]$, $[0, 10]$, $[0, 8]$, $[0, 10]$ and $5, 10, 3, 6, 2, 2$ for methods m_1, m_2, m_3, m_4, m_5, m_6 respectively. The experiments in this section first evaluate the M-DPFP lookahead function and then the complete M-DPFP algorithm.

6.2.1 Efficiency of the M-DPFP Lookahead Function

The first set of experiments involved running the lookahead function for a starting state $s_{0,1}$ in order to determine the total expected utility of an optimal policy for the agents at time $t = 0$. To implement the lookahead function Monte-Carlo sampling was used with sample set sizes between 50 and 400 samples. The first experiment (Figure 6.5) demonstrates how the lookahead function allows for a systematic trade-off of solution quality for speed. In Figure (6.5) the runtime of the lookahead function measured in seconds is plot on the x-axis (notice the logarithmic scale) and the computed total expected utility of an optimal policy from state $s_{0,1}$ is plot on the y-axis. Because the expected utility returned by the lookahead function always underestimates the true expected utility, the y-axis is simply labelled as *Solution quality*. Data points for each size of the Monte-Carlo sample set correspond (from left to right) to $\kappa = 0.3, 0.25, 0.2, 0.15, 0.1$ while each data point is an average of 5 algorithm runs.

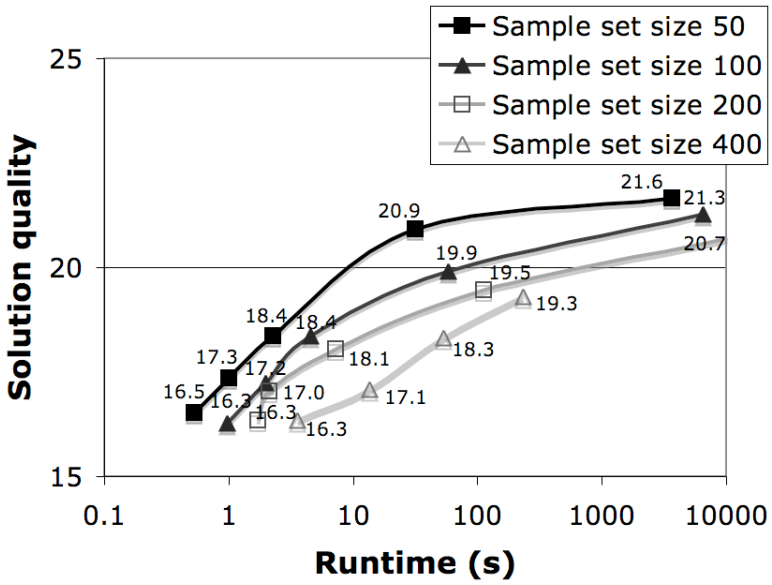

Figure 6.5: Runtime and quality of the lookahead function

First, observe that the solution quality converges as κ decreases and that a linear decrease of κ corresponds to an exponential increase of runtime. Also, an exponential increase of the size of Monte-Carlo sample set surprisingly translates to only linear increase of the algorithm runtime whereas the solution quality is comparable, e.g., for $\kappa = 0.15$ the solution quality for all sizes of sample sets differs by less than 8%. This last result is encouraging, as is allows the lookahead function to find accurate solutions even for small sets of Monte-Carlo samples.

The second experiment shows that the lookahead function finds solutions with quality guarantees. In Figure 6.6 the algorithm runtime is marked on the x-axis and the the solution quality is marked on the y-axis. For $\kappa \in \{0.3, 0.25, 0.2, 0.15, 0.1\}$ the error bound formula 5.20 is used to determine the hypothetical maximal quality of the solution; this quality is then contrasted with the solution quality found by the lookahead function for a fixed size of the Monte-Carlo sample set. Observe that the lookahead function finds solutions with quality guarantees, e.g., if $\kappa = 0.1$,

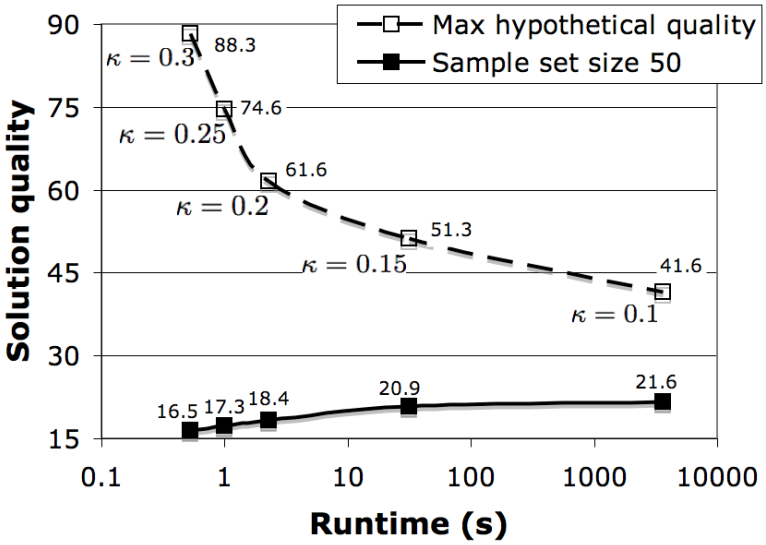

Figure 6.6: Error bound of the of the Lookahead function

it finds a solution that is guaranteed to be less than 50% away from the optimal solution. Also, the lookahead function can trade off speed (by decreasing κ) for optimality.

6.2.2 Efficiency of the M-DPFP algorithm

The second set of experiments reports on the efficiency of the complete M-DPFP algorithm. Here, the Monte-Carlo sample set size is fixed to 50, but the number of representative states is varied, i.e., representative state branching factors of 20, 40 and 80 are used (for the definition of the branching factor, refer to Section 5.2.3.1).

The experiment in Figure 6.7 reports on the total runtime of M-DPFP; the state branching factor is varied in the x-axis and the total runtime of M-DPFP is plot on the y-axis — each data point is an average of 5 algorithm runs. As can be seen, the exponential increase of representative state branching factor translates into the exponential increase of M-DPFP runtime which is not surprising. More interesting is the the proportion of time that M-DPFP spends finding the optimal

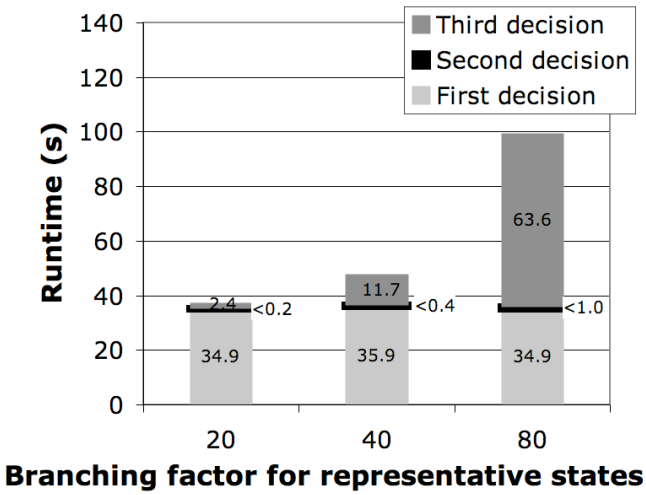

Figure 6.7: Runtime of the M-DPFP algorithm

first decision (the joint action from joint state $(s_{0,1}, s_{0,2})$, the second decision (optimal actions after one method has been executed) and the third decision (optimal actions after two methods have been executed). When the branching factor for representative states is 20, over 92% of the total runtime of M-DPFP is spend on finding the optimal first decision, i.e., the lookahead function for state $(s_{0,1}, s_{0,2})$ is the major bottleneck of the algorithm. However, as the branching factor increases, time spent on finding the optimal later decisions becomes a new bottleneck of the algorithm.

Hence, representative states must be generated wisely, as their number directly affects the runtime of M-DPFP. The final experiment shows how different representative state selection heuristics affect the maximum *quality loss* of M-DPFP. In Figure 6.8 x-axis represents the branching factor for representative states and the y-axis is the maximum quality loss of M-DPFP calculated using formula 5.12. The first thing to notice is that the increase in the branching factor for representative states translates into the decrease of the maximum quality loss of M-DPFP for both the H_{UT} and H_{UP} heuristics. Furthermore, when M-DPFP uses the H_{UT} heuristic, the maximum

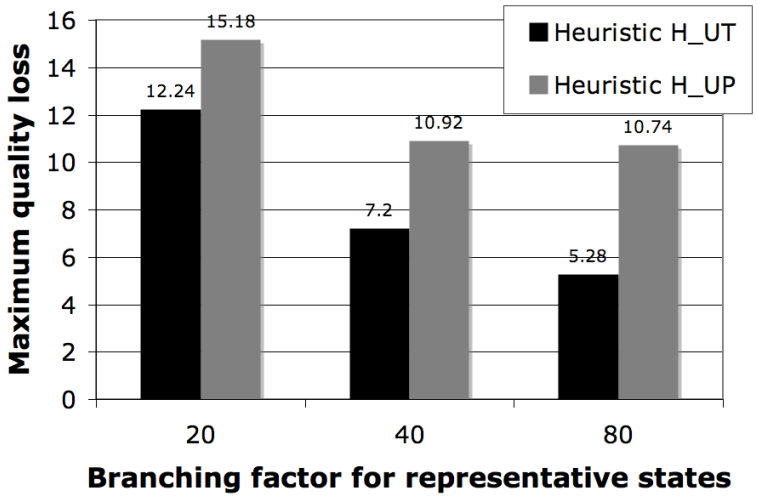

Figure 6.8: Quality loss of the M-DPFP algorithm for different representative state selection heuristics

quality loss is generally smaller than the quality loss when M-DPFP uses the H_{UP} heuristic. This result is in accordance with the intuition that the uniform distribution of representative states over time, generated by the H_{UT} heuristic, guarantees that all the states on the time dimension are well represented. In contrast, H_{UP}'s discrimination in choosing where the representative states should be added, favoring higher-likelihood regions of the time dimension, translates into higher maximum quality loss values. Note however that in practice, M-DPFP running with the H_{UP} heuristic might outperform M-DPFP running with the H_{UT} as H_{UP}'s representative states are more likely to be visited than the H_{UT}'s representative states and thus, M-DPFP running with the H_{UP} heuristic is less likely to resort to the greedy policy described in Section 5.2.3.2. Indeed the study of probabilistic error bounds [Kearns et al., 1999] for both heuristics is a topic worthy of future investigation.

Chapter 7: Related Work

Planning with continuous resources has been a very active area of research, with many efficient algorithms proposed both for single and multi-agent systems.

7.1 Single Agent Systems

Three classes of algorithms have emerged for planning with continuous resources in single agent systems: (i) Constrained MDPs, (ii) semi MDPs and (iii) continuous state MDPs.

7.1.1 Constrained MDPs

For single agent systems, a popular approach for planning with continuous resources is to use the Constraint MDP framework proposed by [Altman, 1999]. Constraint MDPs, which allow to model continuous resource consumption and resource limits can be formulated as either primal or dual linear programs and solved optimally using existing linear programming solvers. The optimal randomized policy [Paruchuri et al., 2004] can then be found in polynomial time whereas for optimal deterministic policy, one can introduce additional binary constraints [Dolgov and Durfee, 2005] and then solve the optimization problem using efficient mixed integer linear programming tools.

Unfortunately, constrained MDP have two serious restrictions: First, they do not incorporate the resource levels in the state description, and as such, they model resource limits as upper bounds on the total expected discounted costs. As a result, the optimal policy for a constrained MDP is not dependent on actual resource levels which can lead to suboptimal results during policy execution. Second, constrained MDPs assume that each action consumes a deterministic amount of resource, and thus, they are not applicable for modeling our domains of interest.

7.1.2 Semi MDPs

In order to model non-deterministic resource consumption, a popular approach is to use the Semi MDP framework [Howard, 1960]. Semi MDPs, which allow resource consumption to follow arbitrary continuous probability density functions, can be solved efficiently using either value iteration [Bellman, 1957] or policy iteration [Howard, 1960] principles. However, similarly to constrained MDPs, the optimal policies for Semi MDPs are invariant of actual resource limits. Even worse, they cannot handle resource limits, and instead resort to discount factors to model the fact that the MDP process whose actions consume resources will run indefinitely. The same problem pertains to the continuous time MDP model (CTMDP) [Howard, 1971] and the Generalized Semi MDP model (GSMDP) [Younes and Simmons, 2004] and [Younes, 2005], variants of Semi MDPs introduced to handle action concurrency and exogenous events. For both models polynomial time algorithms exist, yet these algorithms only return polices that are not indexed by the amount of resources left.

7.1.3 Continuous state MDPs

To eliminate the shortcomings of the Constrained MDPs and Semi MDPs, that is, to allow for continuous resource consumption and policies dependent of resource availability, the common approach is to encapsulate the amount of resources left in MDP state description. As a result, each MDP state becomes a hybrid state that consists of discrete and continuous components — this model is referred to as continuous state MDP, or continuous resource MDP (if continuous component values always increase/decrease). One can then use either value or policy iteration algorithms to solve the underlying planning problems.

Many value iteration algorithms for solving continuous resource MDPs have been proposed. The easiest (but the least efficient) method is to discretize the continuous resources after which the continuous resource MDP becomes a standard MDP with only discrete states. Alternatively, instead of discretizing the continuous resources, one can assume discrete probability distributions which then discretizes the continuous resources automatically [Boyan and Littman, 2000] — Unfortunately, either way, the discretization invalidates solution quality guarantees and results in a combinatorial explosion of the number of states as it becomes more and more fine-grained.

To remedy the problem of combinatorial explosion of the number of states caused by discretizing the continuous resources, [Feng et al., 2004], [Liu and Koenig, 2005] and [Li and Littman, 2005] have suggested to exploit the structure of the problem. In particular, [Feng et al., 2004] exploits the structure of the transition and reward function to dynamically partition the state space into regions where the value function is constant or changes at the same rate. [Li and Littman, 2005] employs a similar principle but unlike [Feng et al., 2004], allows for continuous transition functions. On the other hand [Liu and Koenig, 2005] considers discrete MDPs, but

allows for continuous, one switch utility functions to model risk-sensitive planning problems. Unfortunately, all these techniques can still be inefficient when the discrete component of states is large, causing a combinatorial explosion of the state-space.

Several promising techniques have been proposed to alleviate the combinatorial explosion caused by the discrete component of states. In particular, [Boutilier et al., 1995, 2000; Guestrin et al., 2004; Dolgov and Durfee, 2006] suggests using dynamic Bayesian Networks to factor the discrete component of the states when discrete state variables depend on each other. Modified versions of dynamic programming that operate directly on the factored representation can then be used to solve the underlying planning problems. More recently, [Mausam et al., 2005] have developed a Hybrid AO* (HAO*) algorithm that significantly improves the efficiency of continuous resource MDP solvers. The idea in HAO* is to prioritize node expansion order based on the heuristic estimate of the node value. Furthermore, HAO* is particularly useful when the starting state, minimal resource consumption and resource constraints are given — HAO* can then prune infeasible trajectories and reduce the number of states to be considered to find an optimal policy.

Since both DPFP and HAO* are the algorithms for speeding up existing continuous resource MDP solvers, there are some apparent similarities between them. However, DPFP differs from HAO* in significant ways. The most important difference is that DPFP's forward search is conducted in a dual space of cumulative distribution functions — in particular, DPFP's key novelty is its search of the different splittings of the cumulative distribution functions entering a particular state (e.g. see the splitting of $F((s_0, s_1))$ in Figure 3.4). This difference leads to a novel approach in DPFP's allocation of effort in determining a policy — less effort is spent on regions of the state space reachable with lower likelihood (e.g. in Figure 3.4 more effort is spent on time interval $[t_1, t_3]$ than on time interval $[t_3, t_4]$). While this effort allocation idea in DPFP differs from

HAO*'s reachability analysis, comparing the runtime efficiency of DPFP and HAO* remains an exciting issue for future work. Such a comparison may potentially lead to creation of a hybrid algorithm combining these two approaches.

On the other hand, DPFP's idea to perform a forward search in continuous state-spaces is similar to [Ng et al., 1999] and [Varakantham et al., 2006], but there are important differences. In particular, [Ng et al., 1999] exploits approximate probability density propagation in context of gradient descent algorithms for searching a space of MDP and POMDP stochastic controllers. Furthermore, for POMDPs and Distributed POMDPs, [Varakantham et al., 2005, 2006] use Lagrangian methods to efficiently calculate the admissible probability ranges, to speed up the search for policies. In contrast, DPFP's tailoring to continuous resource MDPs allows it to exploit the underlying dual space of cumulative distribution functions.

Finally, to address the problem of large runtimes common to value iteration algorithms, a popular approach is to perform policy iteration in the space of basis functions approximating the underlying value functions. This approach assumes that a value function has a shape that can be closely approximated with a linear combination of parametrized basis functions [Lagoudakis and Parr, 2003], [Hauskrecht and Kveton, 2004]. Unfortunately, the selection of the correct family of the basis function is problematic; for example [Nikovski and Brand, 2003] uses Gaussian basis functions to allow for efficient convolution operations, [Mahadevan and Maggioni, 2007] uses spectral analysis to construct Proto value functions that exhibit good characteristics in spatial graphs (assumed to be known) whereas [Petrik, 2007] uses Krylov basis functions that provide a good approximation of the underlying discounted reward-to-go matrices. However, all these algorithms choose their family of basis functions *before* the policy iteration starts which can lead to low quality solutions returned by these algorithms.

7.2 Multiagent Systems

Planning with resources in multiagent systems has received a lot of attention in recent years, due to the increasing popularity of multi-agent domains that require agent coordination under resource constraints [Raja and Lesser, 2003], [Becker et al., 2003], [Lesser et al., 2004], [Musliner et al., 2006]. For the purpose of planning with continuous resources in such domains, one could encode the amount of resource left in description of the state using the Decentralized Markov Decision Processes model (DEC-MDPs) [Goldman and Zilberstein, 2003], Multi agent Team Decision Problem with Communication model (COM-MTDP) [Pynadath and Tambe, 2002] or Partially Observable Stochastic Games model (POSGs) [Hansen et al., 2004].

7.2.1 Globally Optimal Solutions

Unfortunately, the problem of solving DEC-MDPs, COM-MTDPs or POSGs optimally has been proven to be NEXP-complete [Bernstein et al., 2000] and hence, more tractable subclasses of these models have been studied extensively. One example of such subclass is the Network Distributed, Partially Observable MDP model (ND-POMDP) [Nair et al., 2005] in which the agents are transition and observation independent, but share a joint reward structure that gives them an incentive to act in a coordinated fashion. In order to solve ND-POMDP optimally [Varakantham et al., 2007] suggested an efficient branch and bound technique combined with an MDP heuristic that provides an upper bound on the value of a joint policy of a subgroup of agents. More recently, [Marecki et al., 2008] proposed an algorithm (FANS) which builds on the algorithm of [Varakantham et al., 2007]. FANS' use of finite state machines for policy representation allows for varying policy expressivity for different agents, according to their needs. That in turn

allows FANS to scale up to domains involving double digit agent numbers and double digit time horizons.

Another important model is the Transition Independent, Decentralized MDP (TI-DEC-MDP) framework [Becker et al., 2003], a subclass of DEC-MDPs. In order to solve TI-DEC-MDPs optimally, [Becker et al., 2003] proposed an algorithm (CSA) that exploits a geometrical representation of the underlying reward structure. For two agent TI-DEC-MDPs, an even more efficient algorithm has been identified [Petrik and Zilberstein, 2007]; the proposed algorithm casts the planning problems as bilinear programs and then uses standard MILP toolkits to solve them optimally. Finally, the assumption about transition independence of TI-DEC-MDP has been relaxed in the Decentralized MDP with Event Driven Interactions model [Becker et al., 2004] in which the agents are allowed to interact with each other at a fixed number of time points. Although [Becker et al., 2004] proved that the new model can be solved optimally with a modified version of the CSA algorithm, the runtime of the proposed algorithm has been shown to be double exponential in the number of time points at which the agents are allowed to interact. In contrast, the CR-DEC-MDP model in this thesis allows the agents to interact at any point in time from a continuous time interval.

7.2.2 Locally Optimal Solutions

Due to problems with the scale-up to big domains that many globally optimal algorithms face, recent years have seen an increasing popularity of locally optimal algorithms. In particular, for solving ND-POMDPs, [Nair et al., 2005] proposed a synergistic approach that combines the strengths of existing distributed constraint optimization algorithms [Modi et al., 2003] with the advantages of dynamic programming techniques, to scale up to domains with hundreds of agents.

On the other hand, for solving Decentralized POMDPs, [Seuken and Zilberstein, 2007] proposed to combine the standard bottom up, value iteration approach with a forward search guided by the portfolio of heuristic, to scale up to domains with large time horizons.

In the context of Decentralized MDPs [Musliner et al., 2006] suggested to use a fast heuristic to estimate the state value, to focus the search on the most promising parts of the state-space. A similar idea was used in the OC-DEC-MDP algorithm [Beynier and Mouaddib, 2005, 2006] that uses the opportunity cost to estimate the state value. The OC-DEC-MDP algorithm is particularly notable, as it has been shown to scale up to domains with hundreds of tasks and double digit time horizons. Additionally, OC-DEC-MDP is unique in its ability to address both temporal constraints and uncertain method execution durations, which is an important factor for real-world domains. However, OC-DEC-MDP is still slow, because similarly to previously mentioned models, it discretizes the resource levels.

To summarize, research in developing algorithms for planning under uncertainty in multi-agent systems has progressed rapidly in recent years. However, the vast majority of proposed algorithms resort to discretization when dealing with continuous resources. That approach in turn has one major drawback: The discretization process invalidates the solution quality guarantees established for these algorithms. To date, only [Benazera, 2007] proposed a multi-agent framework (TI-DEC-HMDP) that allows for optimal planning with continuous resources in a multiagent setting. However, TI-DEC-HMDP's assumption that agents are transition independent can be problematic in many important multiagent domains.

Chapter 8: Conclusions

8.1 Summary

Recent advances in robotics have made aerial, underwater and terrestrial unmanned autonomous vehicles possible. Many domains for which such unmanned vehicles are constructed are uncertain and exhibit inherent continuous characteristics, such as time required to act or other continuous resources at the disposal of the vehicles, e.g. battery power. Therefore, fast construction of efficient plans for agents acting in such domains characterized by constrained and continuous resources has been a major challenge for AI research. Such rich domains can be modeled as Markov decision processes (MDPs) with continuous resources, that can then be solved in order to construct optimal policies for agents acting in these domains.

This thesis addressed two major unresolved problems in continuous resource MDPs. First, they are very difficult to solve and existing algorithms are either fast, but make additional restrictive assumptions about the model, or do not introduce these assumptions but are very inefficient. Second, continuous resource MDP framework is not directly applicable to multi-agent systems and current approaches commonly discretize resource levels or assume deterministic resource consumption which automatically invalidates the formal solution quality guarantees. My thesis addressed these unresolved problems in three key contributions:

- To speed up the search for policies to continuous resource MDPs, I developed an algorithm called CPH. CPH solves the planning problems at hand by first approximating with a desired accuracy the probability distributions over the resource consumptions with phase-type distributions, which use exponential distributions as building blocks. It then uses value iteration to solve the resulting MDPs more efficiently then its closest competitors.

- To improve the anytime performance of CPH and other continuous resource MDP solvers I developed DPFP, an algorithm that solves the planning problems at hand by performing a forward search in the corresponding dual space of cumulative distribution functions. In doing so, DPFP discriminates in its policy generation effort providing only approximate policies for regions of the state-space reachable with low probability yet it bounds the error that such approximation entails.

- For planning with with continuous resources in multi-agent systems a proposed a novel framework called CR-DEC-MDP and developed two algorithms for solving CR-DEC-MDPs: The first algorithm (VFP) emphasizes scalability. It performs a series of policy iterations in order to quickly find a locally optimal policy. In contrast, the second algorithm (M-DPFP) stresses optimality; it allows for a systematic trade-off of solution quality for speed by using the idea of a forward search in a dual space of cumulative distribution functions in a multi-agent setting.

The empirical evaluation of my algorithms revealed a speedup of up to three orders of magnitude when solving single agent planning problems and up to one order of magnitude when solving

multi-agent planning problems. Furthermore, I demonstrated the practical use of the CPH algorithm in a large-scale disaster simulation DEFACTO. In this context, CPH has contributed to a significant improvement in the efficiency of a simulated disaster rescue operation.

8.2 Future Work

In the future one can imagine that agents will be seamlessly integrated with our society: in the offices, in supply chains, in electronic commerce, in the gaming industry and in all our activities. In fact, many breathtaking agent applications are just around the corner: Robotic cars could soon drive without the human intervention; sensor networks could instantly identify and respond to phenomena in the natural environment and software agents can help to fully automate first responders missions.

These exciting future single and multiagent applications emphasize the research challenge of planning with uncertainty and continuous resources, but in a very large-scale environment. In addressing these challenges I envision continuing to build on some of the basic insights developed in my thesis: exploiting agent interaction structure and tailoring the computation to continuous variables, to fundamentally alter the complexity landscape in addressing such large-scale domains.

In the short run, my goal remains to develop new single and multiagent frameworks for reasoning under uncertainty that will (i) scale up the current state of the art by one order of magnitude, e.g. instead of the 10s of agents in agent networks to 100s of such agents and (ii) allow for multiple depleting or replenishing continuous resources. Such reasoning is important for some of the domains mentioned above, including mobile or stationary sensor networks.

In the longer run, it will be crucial to develop adaptive algorithms that tailor themselves or exploit domain structure themselves, autonomously, rather than relying on human developers' insights. And as I pursue this research trajectory, just as I have done in my thesis work, I will strive to balance the dual goals of developing fundamental research with grounding these ideas in concrete domains.

Bibliography

E. Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1999.

S. Asmussen, O. Nerman, and M. Olsson. Fitting phase-type distributions via the em algorithm. *Scandinavian Journal of Statistics*, 23:419–441, 1996.

R. Beard and T. McLain. Multiple UAV cooperative search under collision avoidance and limited range communication constraints. In *IEEE CDC*, pages 25–30, 2003.

R. Becker, S. Zilberstein, V. Lesser, and C. V. Goldman. Transition-Independent Decentralized Markov Decision Processes. In *AAMAS*, pages 41–48, 2003.

R. Becker, V. Lesser, and S. Zilberstein. Decentralized MDPs with Event-Driven Interactions. In *AAMAS*, pages 302–309, 2004.

R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

E. Benazera. Solving decentralized continuous markov decision problems with structured reward. In *Kunstlische Intelligenz*, 2007.

D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of Markov decision processes. In *UAI*, pages 32–37, 2000.

A. Beynier and A. Mouaddib. A polynomial algorithm for decentralized Markov decision processes with temporal constraints. In *AAMAS*, pages 963–969, 2005.

A. Beynier and A. Mouaddib. An iterative algorithm for solving constrained decentralized Markov decision processes. In *AAAI*, pages 1089–1094, 2006.

D. Blidberg. The development of autonomous underwater vehicles (AUVs); a brief summary. In *ICRA*, 2001.

A. Bobbio and A. Cumani. Ml estimation of the parameters of a ph distribution in triangu- lar canonical form. *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 33–46, 1992.

C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In Chris Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1111, San Francisco, 1995. Morgan Kaufmann. URL `citeseer.ist.psu.edu/boutilier95exploiting.html`.

C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000. URL citeseer.ist.psu.edu/boutilier99stochastic.html.

J. Boyan and M. Littman. Exact solutions to time-dependent MDPs. In *NIPS*, pages 1026–1032, 2000.

J. Bresina, R. Dearden, N. Meuleau, D. Smith, and R. Washington. Planning under continuous time and resource uncertainty: A challenge for AI. In *UAI*, 2002.

D.R. Cox. A use of complex probabilities in the theory of stochastic processes. *Proceedings of the Cambridge Philosophical Society 51*, 2:313–319, 1955.

K. Decker and V. Lesser. Designing a Family of Coordination Algorithms. *ICMAS-95*, January 1995. URL http://mas.cs.umass.edu/paper/30.

A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological) 39*, 1:1–38, 1977.

D. A. Dolgov and E. H. Durfee. Stationary deterministic policies for constrained mdps with multiple rewards, costs, and discount factors. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1326–1332, Edinburgh, Scotland, August 2005.

D. A. Dolgov and E. H. Durfee. Symmetric primal-dual approximate linear programming for factored MDPs. In *Proceedings of the Ninth International Symposiums on Artificial Intelligence and Mathematics (AI&M 2006)*, Florida, January 2006.

A.K. Erlang. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *The Post Ofce Electrical Engineers Journal*, 10:189–197, 1917.

Z. Feng, R. Dearden, N. Meuleau, and R. Washington. Dynamic programming for structured continuous MDPs. In *UAI*, 2004.

C. Goldman and S. Zilberstein. Optimizing information exchange in cooperative multi-agent systems, 2003.

C. Guestrin, M. Hauskrecht, and B. Kveton. Solving factored mdps with continuous and discrete variables, 2004.

E. Hansen, D. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), pp. 709–715.*, 2004. URL citeseer.ist.psu.edu/hansen04dynamic.html.

M. Hauskrecht and B. Kveton. Linear programm approximations for factored continuous-state MDPs. In *Advances in Neural Information Processing Systems 16*. MIT Press, 2004.

R.A. Howard. *Dynamic Programming and Markov Processes*. John Wiley and Sons, New York, 1960.

R.A. Howard. *Dynamic Probabilistic Systems*, volume 2. John Wiley and Sons, New York, 1971.

M.A. Johnson. Matching moments to phase distributions: Nonlinear programming approaches. *Communi- cations in Statistics — Stochastic Models 6*, 2:259–281, 1990.

M. J. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *IJCAI*, pages 1324–1231, 1999. URL `citeseer.ist.psu.edu/kearns01sparse.html`.

S. Kullback and R.A. Leibler. On information and sufciency. *Annals of Mathematical Statistics 22*, 1:79–86, 1951.

M. Lagoudakis and R. Parr. Least-squares policy iteration, 2003.

V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X.Q. Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.

L. Li and M. Littman. Lazy approximation for solving continuous finite-horizon MDPs. In *AAAI*, pages 1175–1180, 2005.

Y. Liu and S. Koenig. Risk-sensitive planning with one-switch utility functions: Value iteration. In *AAAI*, pages 993–999, 2005.

D. J. C. MacKay. Introduction to Monte Carlo methods. In M. I. Jordan, editor, *Learning in Graphical Models*, NATO Science Series, pages 175–204. Kluwer Academic Press, 1998.

S. Mahadevan and M. Maggioni. Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes. *Journal of Machine Learning Research*, 8:2169–2231, 2007.

J. Marecki, N. Schurr, M. Tambe, and P. Scerri. Dangers in multiagent rescue using defacto. In *Proceedings of 2nd International Workshop on Safety and Security in Multiagent Systems held at the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.

J. Marecki, S. Koenig, and M.Tambe. A fast analytical algorithm for solving Markov decision processes with real-valued resources. In *IJCAI*, 2007. To Appear.

J. Marecki, T. Gupta, P. Varakantham, M. Tambe, and M. Yokoo. Not all agents are equal: Scaling up distributed pomdps for agent networks. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS-08)*, 2008.

R. Marie. Calculating equilibrium probabilities for $\lambda(n)/c_k/1/n$ queues. In *Proceedings of the 7th IFIP W.G.7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 117–125. ACM SIGMETRICS, 1980.

Mausam, E. Benazera, R. I. Brafman, N. Meuleau, and E. A. Hansen. Planning with continuous resources in stochastic domains. In *IJCAI*, pages 1244–1251, 2005.

P. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS*, 2003. URL citeseer.comp.nus.edu.sg/549918.html.

D. Musliner, E. Durfee, J. Wu, D. Dolgov, R. Goldman, and M. Boddy. Coordinated plan management using multiagent MDPs. In *AAAI Spring Symposium*, 2006.

R. Nair, M. Tambe, M. Yokoo, D. Pynadath, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *IJCAI*, pages 705–711, 2003.

R. Nair, M. Roth, M. Yokoo, and M. Tambe. Communication for improving policy computation in distributed pomdps. In *Proceedings of the Third International Joint Conference on Agents and Multiagent Systems (AAMAS-04)*, pages 1098–1105, 2004. URL citeseer.ist.psu.edu/nair04communication.html.

R. Nair, P. Varakantham, M. Tambe, and M. Yokoo. Networked distributed POMDPs: A synergy of distributed constraint optimization and POMDPs. In *IJCAI*, pages 1758–1760, 2005.

M. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. John Hopkins University Press, 1981.

A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*, pages 278–287. Morgan Kaufmann, San Francisco, CA, 1999. URL citeseer.ist.psu.edu/article/ng99policy.html.

D. Nikovski and M Brand. Non-Linear stochastic control in continuous state spaces by exact integration in Bellman's equations. In *ICAPS*, 2003.

T. Osogami and M. Harchol-Balter. A closed-form solution for mapping general distribu- tions to minimal ph distributions. In *Proceedings of the 13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–217. Springer, 2003.

P. Paruchuri, M. Tambe, F. Ordonez, and S. Kraus. Towards a formalization of teamwork with resource constraints, 2004. URL citeseer.ist.psu.edu/article/paruchuri04towards.html.

L. Pedersen, D.E. Smith, M. Deans, R. Sargent, C. Kunz, D. Lees, and S. Rajagopalan. Mission planning and target tracking for autonomous instrument placement. In *IEEE Aerospace Conference*, pages 34–51, 2005.

M. Petrik. An analysis of laplacian methods for value function approximation in mdps. In *IJCAI*, pages 2574–2579, 2007.

M. Petrik and S. Zilberstein. Anytime coordination using separable bilinear programs. In *AAAI*, 2007.

M. Puterman. *Markov decision processes*. John Wiley and Sons, New York, 1994.

D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. In *JAIR*, volume 16, pages 389–432, 2002.

A. Raja and V. Lesser. Efficient meta-level control in bounded-rational agents. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1104–1105, New York, NY, USA, 2003. ACM Press.

C.H. Sauer and K.M. Chandy. Approximate analysis of central server models. *IBM Journal of Research and Development 19*, 3:301–313, 1975.

N. Schurr, J. Marecki, P. Scerri, J. Lewi, and M. Tambe. The defacto system: Coordinating human-agent teams for the future of disaster response, 2005.

N. Schurr, J. Marecki, and M. Tambe. Riaact: A robust approach to adjustable autonomy for human-multiagent teams. In *Short Paper in Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS-08)*, 2008.

S. Seuken and S. Zilberstein. Memory-bounded dynamic programming for dec-pomdps. In *IJCAI*, 2007.

M. Telek and A. Heindl. Matching moments for acyclic discrete and continuous phase-type distributions of second order. *International Journal of Simulation Systems, Science and Technology 3*, 3–4:47–57, 2002.

S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. Winning the darpa grand challenge. *Journal of Field Robotics*, 2006. accepted for publication.

P. Varakantham, R. Maheswaran, and M. Tambe. Exploiting belief bounds: Practical POMDPs for personal assistant agents. In *Proceedings of AAMAS-05*, pages 978–985, 2005. URL `citeseer.ist.psu.edu/varakantham05exploiting.html`.

P. Varakantham, R. Nair, M. Tambe, and M. Yokoo. Winning back the cup for distributed POMDPs: Planning over continuous belief spaces. In *Proceedings of AAMAS-06*, pages 289–296, May 2006.

P. Varakantham, J. Marecki, M. Tambe, and M. Yokoo. Letting loose a spider on a network of pomdps: Generating quality guaranteed policies. In *Proceedings of AAMAS-07*, May 2007.

H. Younes. Planning and execution with phase transitions. In *AAAI*, 2005.

H. Younes and R. Simmons. Solving generalized semi-MDPs using continuous phase-type distributions. In *AAAI*, pages 742–747, 2004.

Appendix A: Phase Type Distribution

This appendix introduces the formal definition of a phase-type distribution. Let $E(\lambda)$ denote an exponential probability density function given by the formula $E(\lambda)(t) = \lambda e^{-\lambda} t$ where $\lambda > 0$ is the *exit rate* parameter. Also, let $s_1, ..., s_m$ denote the *transitory states (also called phases)* and s_{m+1} denote the *absorbing state* of a Markovian process — the transitory states are not the real states of an MDP, and they are only introduced for the purposes of phase-type approximation. Furthermore, let $\overrightarrow{\alpha} = (p_1, ..., p_m) : \sum_{i=1}^{m} p_i = 1$ specify the initial discrete distribution over transitory states, i.e., a Markovian process starts in transitory state s_i with probability p_i. Finally, let $p_{i,j}$ be the probability that the process will transition from state s_i to state s_j for a transition whose duration follows $E(\lambda_i)$ where λ_i is the exit rate associated with state s_i — for all $i = 1, ..., m$ we have $\sum_{j=1}^{m} p_{i,j} = 1 - q_i$ where q_i is the probability of transitioning to the absorbing state s_{m+1}

from phase s_i. In order to write down the formula for a phase-type distribution, the probabilities $p_{i,j}$ and exit rates λ_i are first encoded in an *infinitesimal generator matrix* Q given by:

$$Q = \begin{pmatrix} -\lambda_1(1-p_{1,1}) & \lambda_1 p_{1,2} & \lambda_1 p_{1,3} & \dots & \lambda_1 p_{1,m-1} & \lambda_1 p_{1,m} \\ \lambda_2 p_{2,1} & -\lambda_2(1-p_{2,2}) & \lambda_2 p_{2,3} & \dots & \lambda_2 p_{2,m-1} & \lambda_2 p_{2,m} \\ \lambda_3 p_{3,1} & \lambda_3 p_{3,2} & -\lambda_3(1-p_{3,3}) & \ddots & \lambda_3 p_{3,m-1} & \lambda_3 p_{3,m} \\ \dots & \dots & \ddots & \ddots & \ddots & \vdots \\ \lambda_{m-1} p_{m-1,1} & \lambda_{m-1} p_{m-1,2} & \lambda_{m-1} p_{m-1,3} & \ddots & -\lambda_{m-1}(1-p_{m-1,m-1}) & \lambda_{m-1} p_{m-1,m} \\ \lambda_m p_{m,1} & \lambda_m p_{m,2} & \lambda_m p_{m,3} & \dots & \lambda_m p_{m,m-1} & -\lambda_m(1-p_{m,m}) \end{pmatrix} \tag{A.1}$$

A Phase-type distribution $f(\overrightarrow{\alpha}, Q)$, which specifies the probabilities of entering the absorbing state s_{m+1} over time, is then given by:

$$f(\overrightarrow{\alpha}, Q)(t) = -\overrightarrow{\alpha} e^{Qt}(-Q\overrightarrow{1}) \tag{A.2}$$

where $\overrightarrow{1}$ is the unit column vector of size m. The corresponding cumulative distribution function is then given by:

$$\int_{-\infty}^{t} f(\overrightarrow{\alpha}, Q)(t')dt' = 1 - \overrightarrow{\alpha} e^{tQ}\overrightarrow{1} \tag{A.3}$$

Appendix B: Classes of Phase Type Distributions

Thanks to their analytical properties, some classes of phase-type distributions have been given particular attention. We now illustrate some of the most important classes and show their generator matrices Q.

- **Exponential distribution**: If $m = 1$, $p_1 = 1$ and $p_{1,1} = 0$ then the phase-type distribution $f(\overrightarrow{\alpha}, Q) = \lambda_1 e^{-\lambda_1 t} = E(\lambda_1)$ is an exponential distribution. Here, the generator matrix Q is simply $Q = (\lambda_1)$ for a given exit rate parameter λ_1.

- **Erlang distribution** [Erlang, 1917] (also known as hyper-exponential distribution) is a phase-type distribution determined by two parameters: the number of phases m and the common exit rate λ. Intuitively, an m-phase Erlang distribution is a deterministic sequence of m exponential distributions, each once sampled from $E(\lambda)$. Formally, $p_1 = 1$, $p_i = 0$, $p_{i,i+1} = 1$ and $\lambda_i = \lambda$ for all $i = 1, ..., m$. The generator matrix Q for the m-phase Erlang distribution is then the following:

$$Q = \begin{pmatrix} -\lambda & \lambda & 0 & 0 & \dots & 0 & 0 \\ 0 & -\lambda & \lambda & 0 & \dots & 0 & 0 \\ 0 & 0 & -\lambda & \lambda & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -\lambda & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -\lambda \end{pmatrix} \tag{B.1}$$

- **Generalized Erlang distribution** is a generalization to the Erlang distribution that allows the process to transition from state s_1 to the absorbing state s_{m+1}, that is, $q_1 \geq 0$. Formally, $p_1 = 1$, $p_i = 0$, $\lambda_i = \lambda$ for all $i = 1, ..., m$ and $p_{i,i+1} = 1$ for all $i = 2, ..., m$. However, $p_{1,2} = p$ and $p_{1,i} = 0$ for $i = 2, ..., m$ for a given probability $p \geq 0$. The generator matrix Q for the m-phase generalized Erlang distribution is then the following:

$$Q = \begin{pmatrix} -\lambda & p\lambda & 0 & 0 & \dots & 0 & 0 \\ 0 & -\lambda & \lambda & 0 & \dots & 0 & 0 \\ 0 & 0 & -\lambda & \lambda & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -\lambda & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & -\lambda \end{pmatrix} \tag{B.2}$$

- **Coxian distribution** [Cox, 1955] is a further generalization of the generalized Erlang distribution that allows the process to transition from each state s_i to the absorbing state s_{m+1}, that is, $q_i \geq 0$ for $i = 1, ..., m$. Also, the transition duration distributions are allowed be be different exponential distributions. Formally, an m-phase Coxian distribution is specified by $2m - 1$ parameters: exit rates $\lambda_1, ..., \lambda_m$ and probabilities $p_{i,i+1}$ for $i = 1, ..., m - 1$. It assumes that $p_1 = 1$ and $p_{i,j} = 0$ for all $i = 1, ..., m$ and $j \in \{1, ..., m\} \setminus i + 1$. The generator matrix Q for the m-phase Coxian distribution is then the following:

$$Q = \begin{pmatrix} -\lambda_1 & p_{1,2}\lambda_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & -\lambda_2 & p_{2,3}\lambda_2 & 0 & \dots & 0 & 0 \\ 0 & 0 & -\lambda_3 & p_{3,4}\lambda_3 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ddots & -\lambda_{m-1} & p_{m-1,m}\lambda_{m-1} \\ 0 & 0 & 0 & 0 & \dots & 0 & -\lambda_m \end{pmatrix} \tag{B.3}$$

Appendix C: Fitting to Phase-Type Distributions

There are two leading approaches for determining the coefficients of the generator matrix Q when fitting to phase-type distributions: the *method of moments* and the *method of shapes*. The method of moments aims to approximate the first few moments of the approximated distribution with a phase-type distribution. The nth moment of a real-valued probability density function $f(x)$ of a random variable X is defined as:

$$\mu_n = E[X^n] = \int_{-\infty}^{\infty} x^n f(x) dx. \tag{C.1}$$

In particular, the mean μ of a distribution is the first moment of this distribution, whereas the variance σ^2 of a distribution can be derived with the help of the first two moments:

$$\sigma^2 = E[(X - \sigma_1)^2] = E[X^2] - 2\mu_1 E[X] + \mu_1^2 = \mu_2 - 2\mu_1^2 + \mu_1^2 = \mu_2 - \mu_1^2. \tag{C.2}$$

Although the first moment of a distribution is easily matched by the mean of this distribution, matching subsequent moments is a non-trivial task. In particular, for a *squared coefficient of variation* defined as $cv^2 := \sigma^2/\mu$, we match the first two moments of a distribution depending on the value of cv^2. If $cv^2 < 1$ we use the n-phase generalized Erlang distribution with $n = \lceil 1/(cv^2) \rceil$,

$\lambda = (1 - p + np)/\mu_1$ and $p = 1 - (2n \cdot cv^2 + n - 2 - \sqrt{n^2 + 4 - 4n \cdot cv^2})/(2(n-1)(cv^2+1))$ [Sauer and Chandy, 1975; Marie, 1980]. If, on the other hand, $cv^2 \geq 1/2$ we use a 2-phase Coxian distribution with $\lambda_1 = 2/\mu_1$, $\lambda_2 = 1/(\mu_1 \cdot cv^2)$ and $p = 1/(2 \cdot cv^2)$ [Marie, 1980]. Depending on the values of cv^2 and μ_3, there are also several approaches for matching the first three moments of a distribution [Johnson, 1990; Telek and Heindl, 2002; Osogami and Harchol-Balter, 2003].

Matching the first few moments of an initial distribution does not necessary lead to a good approximation. For example, if a distribution is multi-modal and we match its two first moments, i.e., its mean and standard deviation, then the obtained approximation is still single-modal. In situation like these, it is better to approximate directly the shape of the initial distribution, by minimizing the distance between the initial distribution and its approximation. A common metric for determining the distance between two distributions f and g is the *Kullback-Leibler divergence* (KL-divergence) proposed in [Kullback and Leibler, 1951]. For two probability density functions f and g, their KL-divergence $D_{KL}(f||g)$ is defined as:

$$D_{KL}(f||g) = \int_{-\infty}^{\infty} f(x) \log \frac{f(x)}{g(x)} dx. \tag{C.3}$$

The leading algorithms that approximate the initial distribution by minimizing the KL divergence are the EM (Expectation-Maximization) algorithm [Dempster et al., 1977] and the maximum likelihood estimation algorithm [Bobbio and Cumani, 1992]. Both algorithms take as an input the desired number of phases used by the phase-type distribution as well as the desired structure of the phase-type distribution (e.g. Erlang, Coxian, etc.). In my implementation of CPH I have used the EM algorithm, thanks to its superior convergence properties [Asmussen et al., 1996].

Appendix D: Uniformization of Phase Type Distributions

As shown in Equation A.3, a phase-type distribution in uniquely specified by its generator matrix Q. This matrix, in its most general form, does not make any assumption about the exit rate parameters λ_i which may be arbitrary. However, for many analytical operations, such as the ones performed by CPH, it is required that these exit rates are uniform, that is, $\lambda_1 = ... = \lambda_m = \lambda$. We now show a technique, called *uniformization* [Puterman, 1994], which transforms an arbitrary phase-type distribution to a phase-type distribution with uniform exit rates, that is, $\lambda_1 = ... = \lambda_m = \max_{i=1,...,m} \lambda_i = \lambda$.

For each row j of the generator matrix Q let $k_j := (\max_{i=1,...,m} \lambda_i)/\lambda_j$. Observe, that the generator matrix Q shown in Equation A.1 can also be written as:

$$Q = \begin{pmatrix} -(\lambda_1 k_1)(1 - \frac{k_1 - 1 + p_{1,1}}{k_1}) & (\lambda_1 k_1)\frac{p_{1,2}}{k_1} & \dots & (\lambda_1 k_1)\frac{p_{1,m}}{k_1} \\ (\lambda_2 k_2)\frac{p_{2,1}}{k_2} & -(\lambda_2 k_2)(1 - \frac{k_2 - 1 + p_{2,2}}{k_2}) & \ddots & (\lambda_2 k_2)\frac{p_{2,m}}{k_2} \\ (\lambda_3 k_3)\frac{p_{3,1}}{k_3} & (\lambda_3 k_3)\frac{p_{3,2}}{k_3} & \ddots & (\lambda_3 k_3)\frac{p_{3,m}}{k_3} \\ \dots & \dots & \ddots & \vdots \\ (\lambda_{m-1} k_{m-1})\frac{p_{m-1,1}}{k_{m-1}} & (\lambda_{m-1} k_{m-1})\frac{p_{m-1,2}}{k_{m-1}} & \ddots & (\lambda_{m-1} k_{m-1})\frac{p_{m-1,m}}{k_{m-1}} \\ (\lambda_m k_m)\frac{p_{m,1}}{k_m} & (\lambda_m k_m)\frac{p_{m,2}}{k_m} & \dots & -(\lambda_m k_m)(1 - \frac{k_m - 1 + p_{m,m}}{k_m}) \end{pmatrix} \quad \text{(D.1)}$$

Or simply as:

$$Q = \begin{pmatrix} -\lambda'_1(1-p'_{1,1}) & \lambda'_1 p'_{1,2} & \cdots & \lambda'_1 p'_{1,m} \\ \lambda'_2 p'_{2,1} & -\lambda'_2(1-p'_{2,2}) & \ddots & \lambda'_2 p'_{2,m} \\ \lambda'_3 p'_{3,1} & \lambda'_3 p'_{3,2} & \ddots & \lambda'_3 p'_{3,m} \\ \cdots & \cdots & \ddots & \vdots \\ \lambda'_{m-1} p'_{m-1,1} & \lambda'_{m-1} p'_{m-1,2} & \ddots & \lambda'_{m-1} p'_{m-1,m} \\ \lambda'_m p'_{m,1} & \lambda'_m p'_{m,2} & \cdots & -\lambda'_m(1-p'_{m,m}) \end{pmatrix} \tag{D.2}$$

where $\lambda'_i := \lambda_i k_i$, $p'_{i,j} := \frac{p_{i,j}}{k_i}$ for $i \neq j$ and $p'_{i,i} := \frac{k_i - 1 + p_{i,i}}{k_i}$ are exit rates and state-to-state transition probabilities of a new phase-type distribution $f(\alpha, Q)$. Note that the new phase-type distribution has $\lambda'_i = \max_{i=1,...,m} \lambda_i = \lambda$ for all $i = 1, ..., m$ and thus, the new phase-type distribution is uniformized.